

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

大规模Linux 集群架构最佳实践

如何管理上千台服务器

魔软运维社 著

Best Practice of Managing
Massive Linux Cluster

- 由5位来自著名游戏制造商的Linux技术专家撰写，他们拥有丰富的互联网从业经验，本书是其多年实战经验的精华集合。
- 从实际生产系统和应用出发，以运维的实际工作内容为基本立足点，全方位、真实地展示大规模Linux集群的应用现状。



“集群”属于一门多种技术融合的科学，包含了Linux基础系统、系统安全、系统调优、网络安全、日志分析、系统监控、自动化管理、资产管理等多方面的内容，单人写作很难达到这么全面的剖析范围。本书集结了5位作者的实战经验，他们在各自领域都有深厚的功底，书中内容以笔者团队的日常工作为背景，针对Linux系统、网络、安全、监控、备份、日志分析等内容进行了细致分析，跳出一般书籍仅仅能覆盖的原理层面，详尽真实地展现了各项技术在集群架构和运维方向上的实际应用和发展趋势，其中很多内容更是作者团队多年运维总结的最佳实践。

大规模Linux 集群架构最佳实践

如何管理上千台服务器

Best Practice of Managing
Massive Linux Cluster

魔软运维社 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

大规模 Linux 集群架构最佳实践：如何管理上千台服务器 / 魔软运维社著. —北京：机械工业出版社，2017.8

(Linux/Unix 技术丛书)

ISBN 978-7-111-57585-6

I. 大… II. 魔… III. Linux 操作系统 IV. TP316.85

中国版本图书馆 CIP 数据核字 (2017) 第 188609 号

大规模 Linux 集群架构最佳实践：如何管理上千台服务器

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：缪 杰

责任校对：李秋荣

印 刷：北京文昌阁彩色印刷有限责任公司

版 次：2017 年 9 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16

印 张：25.75

书 号：ISBN 978-7-111-57585-6

定 价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88379426 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

为什么要写这本书

五个 Linux 爱好者和开源软件的密集使用者因为同事关系相聚在动视暴雪，茶余之际谈及目前市场上已出版的 Linux 图书，一致的看法是，虽然市场上以 Linux 为主题的书很多，但绝大多数集中于 Linux 基础介绍或是单纯的服务搭建，有一些书着眼点在 Linux 集群的架构设计，但是往往内容重合度较高、篇幅零散，且基本上限于对原理的讲解，缺乏对实际系统的集成梳理。虽然 Linux 及 Linux 集群目前在互联网已经非常流行，但是基于实际生产应用讲解 Linux 集群的书仍难觅踪迹。因为从严格意义上来说，“集群”属于一门多种技术融合的科学，包含了 Linux 基础系统、系统安全、系统调优、网络安全、日志分析、系统监控、自动化管理、资产管理等多方面的内容，单个人写作很难达到这么全面的剖析范围。于是，我们五人决定合作来写一本相对更全面实用的 Linux 图书。

在决定动笔之际，参与本书写作的五位作者都就职于世界最大的游戏出版公司动视暴雪，因此，本书以动视暴雪中国运维团队的日常工作为背景，内容也基于（但不拘泥）日常运维的生产系统和测试系统，力图从实际生产系统 and 应用出发，以自己平日的实际运维工作为基本立足点，全方位、真实地展示目前 Linux 集群的应用现状。书中内容包括 Linux 系统、网络、安全、监控、备份、日志分析等，跳出了一般书籍仅仅能覆盖的原理层面，详尽真实地展现了各项技术在集群架构和运维方向上的实际应用和发展趋势，其中很多内容更是动视暴雪中国运维团队多年运维总结的最佳实践。

对于我们自己来说，完成这本书的写作，不但能分享自己多年的工作心得，也是一次极为难得的和众多 Linux 爱好者一起学习和成长的机会。

读者对象

本书主要适合于以下读者：

- ☐ 希望更深入地了解 Linux 系统的中高级人员
- ☐ 希望更深入地了解网络的中高级人员

- 基于 Linux 系统的网站前后端开发人员
- 系统运维工程师和架构师

如何阅读本书

本书第 1 章详细描述了 Linux 的安装、配置、用户管理、文件管理、网络管理、进程管理、软件管理等内容，这是 Linux 的基础入门知识，建议所有没有 Linux 基础的读者，或是新手通读本章。第 2 章是 Linux 性能分析，介绍了 Linux 系统中性能分析工具的使用方法，这在实际工作中很常用，但是根据不同的场景，也有很多组合的使用方式。第 3 章至第 5 章是所有生产环境都会使用到的用户集中认证、DNS 服务和系统备份等内容，这些内容属于必知必会的部分，建议通读。第 6 章针对集群和集群存储进行了讲解，建议读者视自己的实际使用情况选读。第 7 章详细介绍了一款当前非常流行的、实时 metric 工具 Graphite，对于很多大型系统来说，这是一款极好的系统状态记录工具。第 8 章介绍 Cobbler，对于依然在使用传统 DC 的管理员来说，Cobbler 是一款很好的系统自动安装配置工具。第 9 章和第 10 章详细描述了 Puppet 在自动化部署中的使用，这也是当前非常流行的一款配置管理工具。第 11 章介绍了 CMDB，建议感兴趣的读者阅读。第 12 章是日志管理内容，描述了两种当前流行的日志处理工具 Splunk 和 ELK，它们都是处理海量日志非常好的工具。

勘误和支持

由于作者水平有限，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正，您有任何宝贵意见都可以发送至邮箱 johnwang.wangjun@gmail.com，我们很期待能够听到您的真挚反馈。

致谢

这本书能顺利的交稿，首先要感谢参与写作的各位作者，能从百忙的工作和各自的家庭生活中抽出宝贵的时间，分享自己的心得和体会，才能有机会让更多的爱好者和同行沟通交流。

此外，感谢机械工业出版社华章公司的编辑杨绣国（Lisa）老师，感谢她在这段时间里始终支持我们的写作，她的鼓励和帮助引导我们顺利完成全部书稿。

王军

2017 年 5 月

Contents 目 录

前言

第 1 章 Linux 系统管理入门.....1

- 1.1 系统安装1
 - 1.1.1 安装 CentOS1
 - 1.1.2 首次启动 CentOS14
 - 1.1.3 更多设置14
- 1.2 系统登录20
 - 1.2.1 本地登录20
 - 1.2.2 远程登录22
- 1.3 用户管理24
 - 1.3.1 用户和用户组的概念24
 - 1.3.2 新增和删除用户25
 - 1.3.3 新增和删除用户组26
 - 1.3.4 用户切换26
- 1.4 文件系统27
 - 1.4.1 什么是文件系统27
 - 1.4.2 常见的文件系统27
 - 1.4.3 磁盘分区和创建文件
系统28
- 1.5 文件管理33
 - 1.5.1 文件和目录简介33
 - 1.5.2 文件和目录权限34
 - 1.5.3 文件查找35

- 1.5.4 文件压缩和打包36
- 1.6 网络管理37
 - 1.6.1 网络配置管理37
 - 1.6.2 Linux 防火墙38
 - 1.6.3 网络连通性诊断40
- 1.7 进程管理42
 - 1.7.1 什么是进程43
 - 1.7.2 进程的常见状态43
 - 1.7.3 进程优先级的调整43
 - 1.7.4 进程的终止44
- 1.8 软件安装46
 - 1.8.1 源码编译安装46
 - 1.8.2 使用包管理 Yum48
 - 1.8.3 创建自己的 Yum 仓库49
- 1.9 系统安全检测与审计51
 - 1.9.1 AIDE 系统入侵检测51
 - 1.9.2 审计53

第 2 章 系统性能分析56

- 2.1 性能分析简介56
- 2.2 系统分析的基本工具56
 - 2.2.1 CPU 性能分析工具56
 - 2.2.2 内存性能分析工具60

2.2.3 磁盘性能分析工具.....	62	4.2.6 利用 DNS 实现负载均衡.....	103
2.2.4 sar.....	64	4.3 DNS 的主从复制.....	104
2.3 软件分析的基本工具.....	66	4.4 配置纯缓存的 DNS 服务.....	106
2.3.1 ldd.....	66	4.5 DNS 的客户端配置.....	107
2.3.2 strace 与 ltrace.....	66	4.5.1 Linux 中的配置.....	107
2.3.3 ipcs.....	71	4.5.2 Windows 中的配置.....	108
2.3.4 systemtap.....	73		
2.4 与内存相关的那些事情.....	76	第 5 章 系统备份	109
2.4.1 内存泄漏.....	76	5.1 为什么要备份.....	109
2.4.2 虚拟内存、物理内存与页缺失.....	78	5.2 常见的备份机制.....	110
2.4.3 Out of Memory.....	79	5.2.1 完全备份.....	110
2.4.4 Overcommit.....	79	5.2.2 增量备份.....	110
2.4.5 cache 与 buffer.....	80	5.2.3 差异备份.....	111
2.5 与磁盘相关的那些事情.....	80	5.3 Bacula 简介.....	111
2.5.1 HDD 与 SSD.....	80	5.3.1 什么是 Bacula.....	111
2.5.2 HDD 磁盘的调度算法.....	81	5.3.2 Bacula 的基本组件.....	112
2.5.3 文件系统日志.....	82	5.4 Bacula 的安装和配置.....	112
2.6 系统资源限制.....	82	5.4.1 Bacula 控制器.....	114
2.6.1 ulimit.....	82	5.4.2 Bacula 存储守护进程.....	120
2.6.2 Cgroup.....	84	5.4.3 Bacula 客户端文件守护进程.....	121
		5.4.4 Bacula 控制台.....	122
第 3 章 用户集中认证	91	5.4.5 启动服务.....	122
3.1 openLDAP 简介.....	91	5.4.6 Bacula 配置综述.....	122
3.2 openLDAP 的安装.....	91	5.5 使用 Bacula 进行备份和恢复.....	124
3.3 openLDAP 的配置.....	92	5.5.1 执行备份.....	124
3.4 利用 openLDAP 集中认证.....	95	5.5.2 文件恢复.....	127
第 4 章 域名服务器 DNS	97	5.6 Bacula 的使用和维护.....	129
4.1 DNS 服务简介.....	97	5.6.1 Bconsole 的用法.....	129
4.2 DNS 安装配置.....	98	5.6.2 使用 Bacula 进行文件验证.....	130
4.2.1 DNS 安装过程.....	98	5.6.3 Catalog 的维护和备份.....	131
4.2.2 关于 chroot 的解释.....	99	5.7 备份的策略.....	132
4.2.3 配置主配置文件.....	99	5.7.1 备份什么.....	133
4.2.4 DNS 的正向解析配置.....	100	5.7.2 备份到哪里.....	133
4.2.5 DNS 的反向解析配置.....	101	5.7.3 备份的时间.....	133
		5.7.4 测试和监控备份.....	133

第6章 集群与存储 134

- 6.1 存储的基本概念 134
- 6.2 SAN 134
 - 6.2.1 SAN 的选择 135
 - 6.2.2 iSCSI 的配置 135
- 6.3 分布式文件系统与集群文件系统 138
 - 6.3.1 分布式文件系统 138
 - 6.3.2 GlusterFS 的配置 138
- 6.4 高可用集群 141
 - 6.4.1 Red Hat HA Cluster 简介 141
 - 6.4.2 配置一个高可用的 Apache 集群 142
- 6.5 负载均衡集群 151
 - 6.5.1 HAProxy 负载均衡 151
 - 6.5.2 Nginx 负载均衡 153
 - 6.5.3 LVS 负载均衡 155

第7章 Graphite 159

- 7.1 Graphite 是什么 159
 - 7.1.1 Graphite 不是一个告警系统 159
 - 7.1.2 Graphite 的功能和特色 159
- 7.2 Graphite 的基本组件 160
 - 7.2.1 Whisper 160
 - 7.2.2 Carbon 161
 - 7.2.3 Graphite Web 162
- 7.3 Graphite 的安装 162
 - 7.3.1 安装 Whisper 数据库 163
 - 7.3.2 安装 Carbon 守护进程 163
 - 7.3.3 安装 graphite-web 163
- 7.4 Graphite 的配置（单点） 164
 - 7.4.1 配置 Carbon 守护进程 164
 - 7.4.2 给 Carbon Cache 发送数据 166
 - 7.4.3 配置 Graphite-web 167

7.5 Graphite 的配置（集群配置） 169

- 7.5.1 配置 Carbon Relay 170
- 7.5.2 Relay 中的数据复制 172
- 7.5.3 数据聚合 172
- 7.5.4 Graphite Cluster 174

7.6 使用 Graphite Web 175

- 7.6.1 Graphite 的 Render API 175
- 7.6.2 Graphite 作图函数 176
- 7.6.3 Graphite Dashboard 和 Grafana 178

7.7 Graphite 的性能监控和调整 181

7.8 其他 182

- 7.8.1 Whisper 文件操作 182
- 7.8.2 压力测试 183
- 7.8.3 其他工具 185

第8章 系统大规模部署 186

8.1 概述 186

8.2 与 PXE 不得不说的故事 186

- 8.2.1 PXE 简介 186
- 8.2.2 PXE 实战 187

8.3 系统部署工具 Cobbler 192

- 8.3.1 Cobbler 简介 192
- 8.3.2 Cobbler 安装 192
- 8.3.3 Cobbler 配置 193
- 8.3.4 Cobbler 应用 197
- 8.3.5 Cobbler API 202
- 8.3.6 Cobbler Replication 203
- 8.3.7 Cobbler 实战 204

8.4 操作系统无盘技术 206

- 8.4.1 定义 206
- 8.4.2 制作无盘镜像 206
- 8.4.3 测试无盘镜像 212

8.5 本章小结 213

第 9 章 Puppet 配置管理214	第 11 章 CMDB 配置中心管理314
9.1 什么是 Puppet..... 214	11.1 什么是 DCIM..... 314
9.1.1 Puppet 对于系统运维意味 着什么..... 214	11.2 什么是 CMDB..... 315
9.1.2 为什么选择 Puppet..... 215	11.3 运维为什么需要 CMDB..... 316
9.2 安装 Puppet..... 216	11.3.1 整合信息..... 316
9.2.1 准备工作..... 216	11.3.2 关系映射..... 316
9.2.2 安装一个服务端..... 219	11.3.3 防止配置偏差..... 316
9.2.3 安装一个客户端..... 219	11.3.4 自动化..... 317
9.2.4 连接第一个客户端..... 220	11.3.5 中央管理..... 317
9.2.5 Puppet master 上的 site.pp..... 220	11.4 如何选择适合的 CMDB..... 317
9.2.6 制作第一个模块..... 223	11.4.1 每个项目都会遇到的那些 任务..... 317
9.3 深入 Puppet..... 227	11.4.2 选择开源的 CMDB..... 321
9.3.1 深入 resources type..... 227	11.5 自主搭建 CMDB..... 324
9.3.2 深入 metaparameter..... 240	11.5.1 openDCIM 安装..... 324
9.3.3 深入 fact..... 245	11.5.2 openDCIM 配置..... 327
9.3.4 深入流程控制..... 248	11.5.3 openDCIM API..... 339
9.3.5 深入 function..... 252	11.5.4 解决每个项目都会遇到的 那些任务..... 359
9.3.6 深入 template..... 257	11.6 如何管理好一个 CMDB..... 371
9.3.7 深入 define type..... 259	11.6.1 制定相应流程管理..... 371
第 10 章 Puppet 实战262	11.6.2 CMDB 与自动化..... 373
10.1 扩展 Puppet..... 262	11.6.3 做好 CMDB 的架构设计..... 374
10.1.1 自定义模块..... 262	11.6.4 那些年,我们碰过的坑..... 375
10.1.2 使用公有模块..... 271	第 12 章 日志管理378
10.1.3 神奇的 enc..... 273	12.1 日志中的四个 W..... 378
10.1.4 自定义 resource type/facter/ function..... 275	12.2 首先要有一个日志服务器..... 378
10.2 管理好一个 Puppet 集群..... 280	12.2.1 rsyslog..... 379
10.2.1 监控 Puppet 运行状况..... 280	12.2.2 syslog-ng..... 380
10.2.2 做好 Puppet 的容量规划..... 288	12.2.3 如何选择 syslog 程序..... 382
10.2.3 使用版本控制来管理代码..... 295	12.3 常见的日志分析处理工具..... 382
10.2.4 确保你的代码不是留给别人 的坑..... 311	12.4 Splunk 的安装配置..... 384
	12.4.1 下载 Splunk 安装程序包..... 384

12.4.2	安装启动 Splunk	384	12.5.4	配置 Elasticsearch	392
12.4.3	配置 Splunk	385	12.5.5	配置 Kibana	393
12.4.4	搜索日志	388	12.6	Elasticsearch 入门	395
12.5	Elasticsearch+Logstash+Kiana	388	12.6.1	基本配置	395
12.5.1	ELK 简介	388	12.6.2	安装插件	397
12.5.2	安装 ELK 软件包	389	12.6.3	API	397
12.5.3	配置 Logstash	391			

Linux 系统管理入门

1.1 系统安装

据不完全统计，目前世界上有大概 300 多种 Linux 发行版，选择什么样的 Linux 发行版成为安装前的第一个问题。在众多发行版中，RedHat 作为一个成熟的商用发行版，不仅经过了多年的市场考验，也有成熟的认证体系，最重要的是有活跃的读者社区，所以对于初学者而言，RedHat 无疑是最好的选择。不过，因其“商用”背景，在使用 RedHat 时会有一些细节上的限制。近年来，另一个 Linux 的重要发行版 CentOS 的发展极为迅速，这个发行版的版本发布和 RedHat 保持一致，在使用上几乎完全相同，在本书动笔之时 CentOS 最新的版本已经是 7，但是由于 CentOS 5/6 目前使用者众多，所以本书将以 CentOS 6.6 作为演示，读者可以使用虚拟机进行学习和测试。

工欲善其事，必先利其器，本章将开门见山、直奔主题，下面会使用过程截图为大家演示 Linux 系统的具体安装步骤。

1.1.1 安装 CentOS

安装 CentOS 首先需要获得发行版的安装介质，可以通过 www.centos.org 下载（如图 1-1 所示），为了获取最快的下载速度，读者可以选择离自己比较近的镜像站点。

下载完成后，如果需要在物理机上安装，则需要将该镜像烧制成可启动的 CD，并设置计算机的启动设备为 CD。如果是使用虚拟机安装，也需要进行相关的设置。这里笔者将使用 VMware Workstation 进行演示。

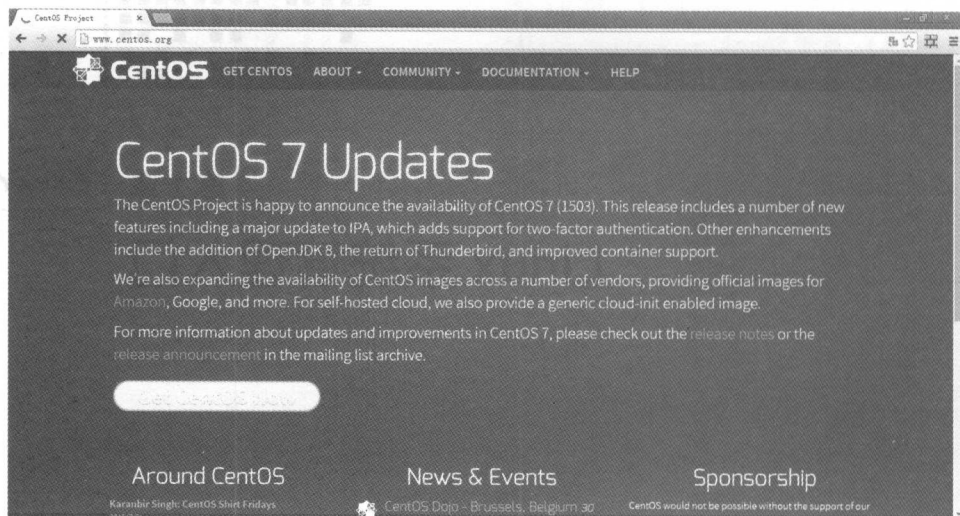


图 1-1 下载 CentOS

打开 VMware Workstation 软件并选择“创建新的虚拟机”(如图 1-2 所示)。

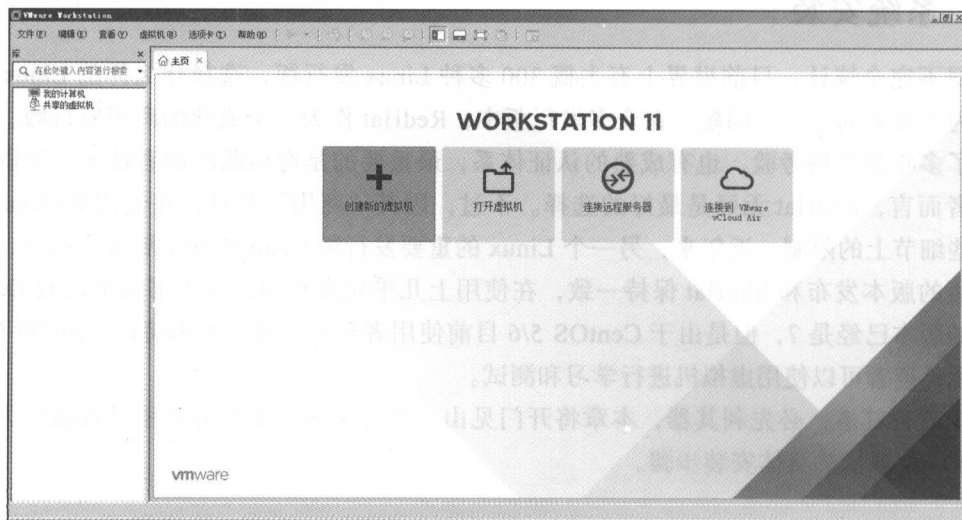


图 1-2 创建新的虚拟机

在随后出现的“新建虚拟机向导”中，入门安装推荐选择“典型”(如图 1-3 所示)。

在“安装客户机操作系统”页面，选择“稍后安装操作系统”(如图 1-4 所示)。

在“选择客户机操作系统”页面中(如图 1-5 所示)，选择“Linux”并在版本中选择“CentOS 64 位”。

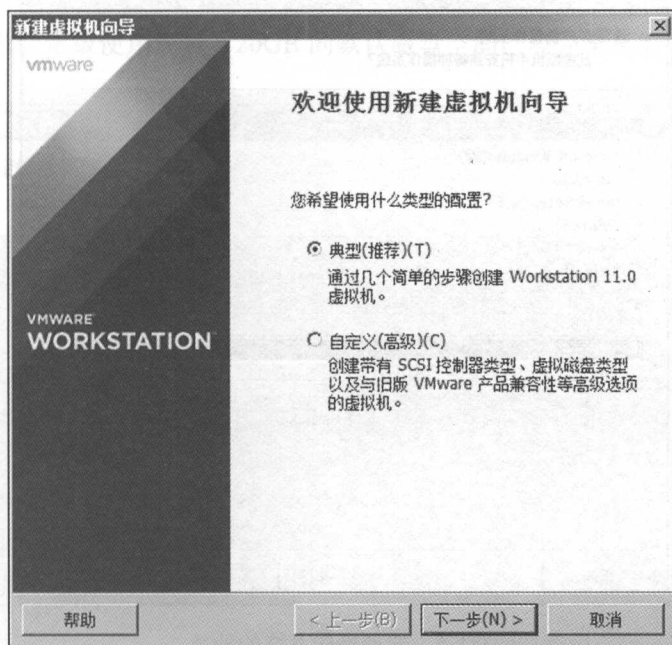


图 1-3 使用“典型”方式创建虚拟机

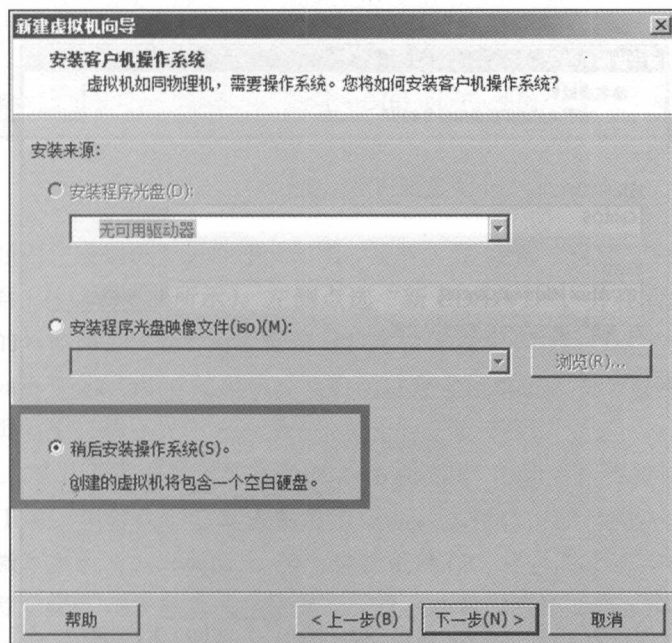


图 1-4 选择“稍后安装操作系统”

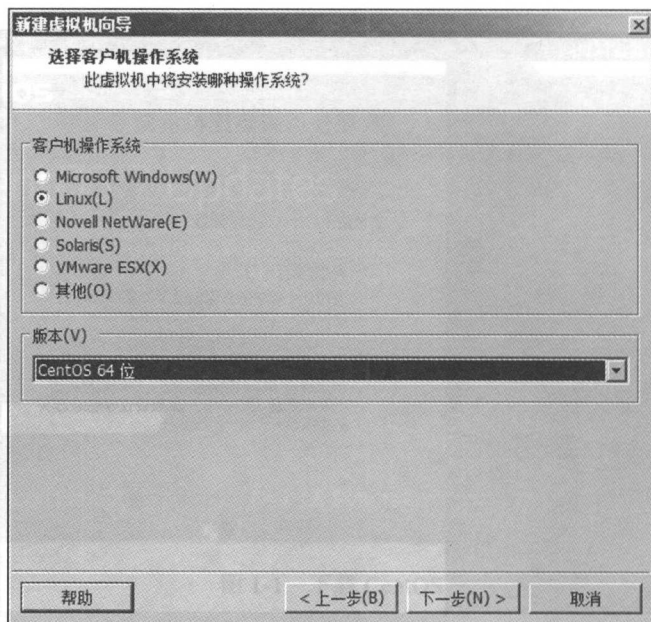


图 1-5 选择操作系统的种类

在“命名虚拟机”页面中(如图 1-6 所示),给虚拟机起一个名字,并选择存储路径。读者不必拘泥于本书介绍,根据自身实际情况设置即可。

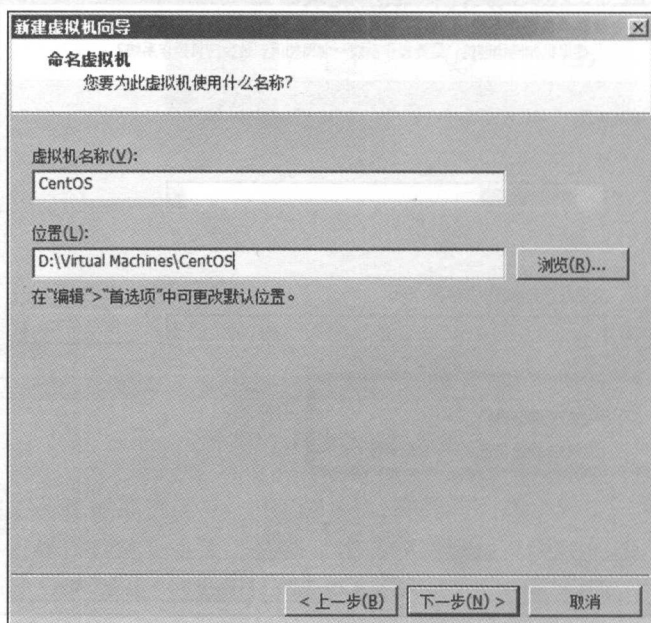


图 1-6 选择虚拟机存储路径

在“指定磁盘容量”页面中（如图 1-7 所示），读者可以自行调整虚拟机磁盘的大小。作为初学或大多数轻量级使用而言，20GB 的默认磁盘空间已经完全足够。

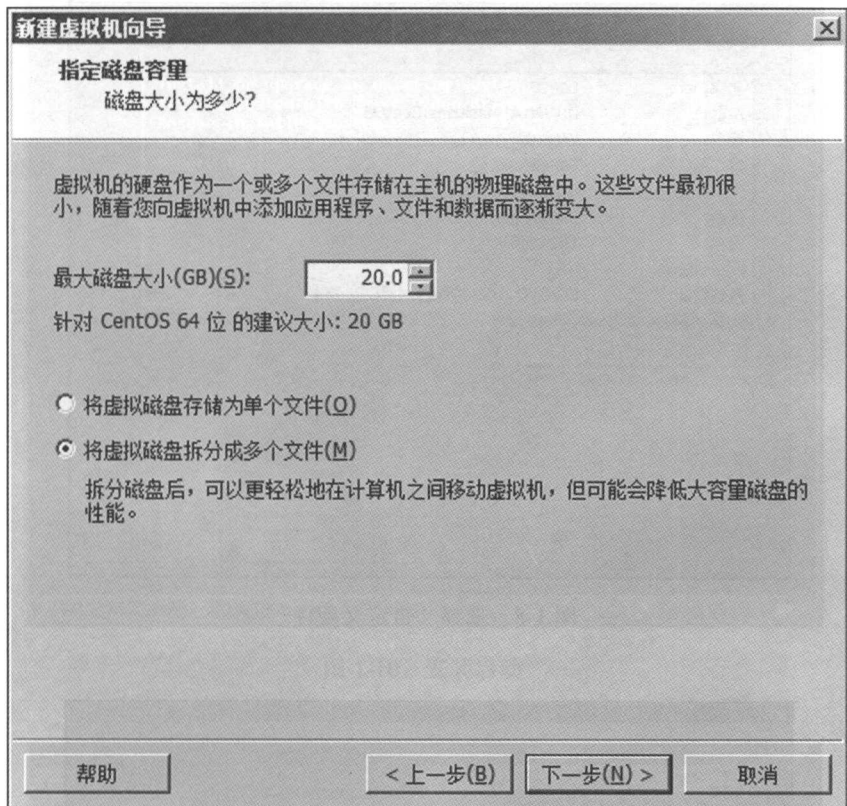


图 1-7 设置虚拟机磁盘大小

在“已准备好创建虚拟机”页面中（如图 1-8 所示），点选“自定义硬件”。并在随后弹出的“硬件”页面中（如图 1-9 所示），左侧点选“新 CD/DVD”，并在右侧指定之前下载到的 ISO 镜像文件的具体路径（读者请根据自身实际情况设置）随后点选“关闭”完成最终设置，最后在 VMware Workstation 的起始页面启动这台虚拟机进入安装过程。

在“硬件”页面中，选择光驱并选择 CentOS 的安装镜像。

机器启动后，便进入了安装过程（如图 1-10 所示），启动后选择第一项或是第二项均可，区别主要在于第二项将会安装基本的显卡驱动。选择后，回车确认。机器将首先载入一个安装系统的微型系统（anaconda），然后会尝试检查安装介质是否存在问题影响实际安装，当然如果读者下载到 ISO 后确认完整无误，这一步可以省略（如图 1-11 所示）。

剩下的安装步骤，请读者参阅图 1-12～图 1-24 进行。

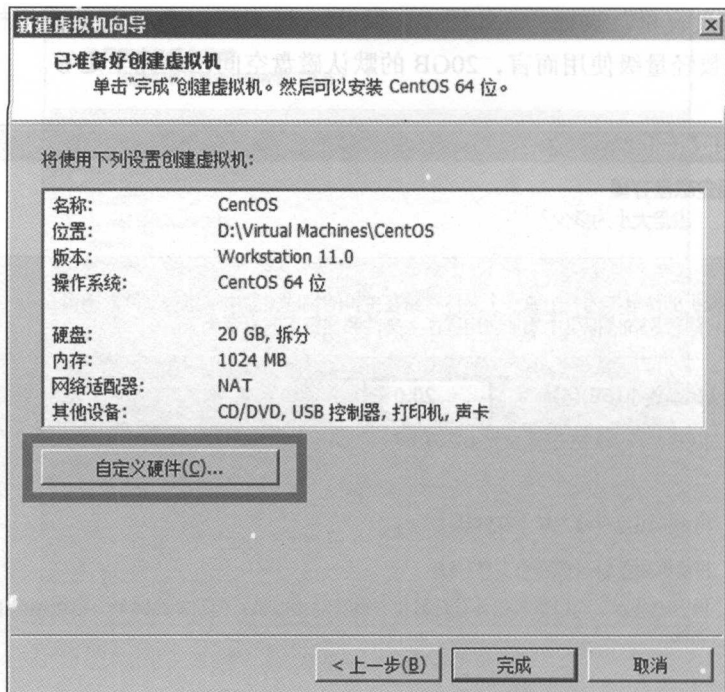


图 1-8 选择“自定义硬件”

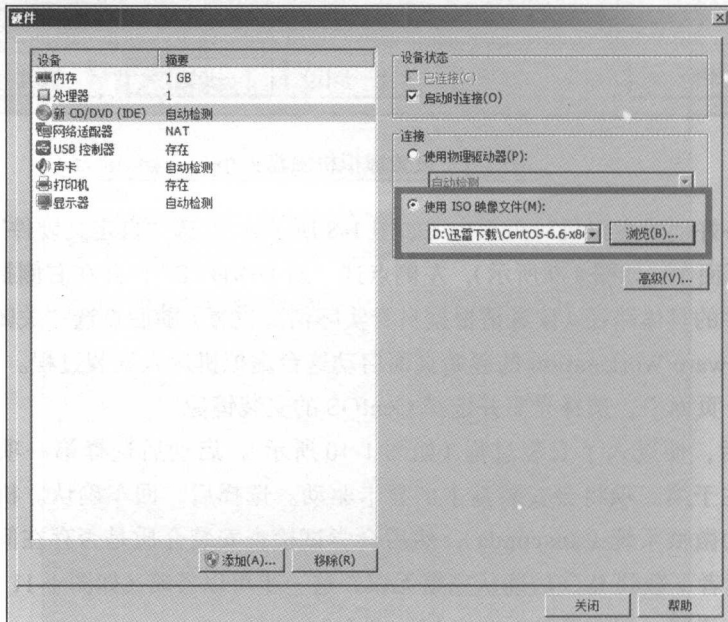


图 1-9 指定 ISO 镜像地址

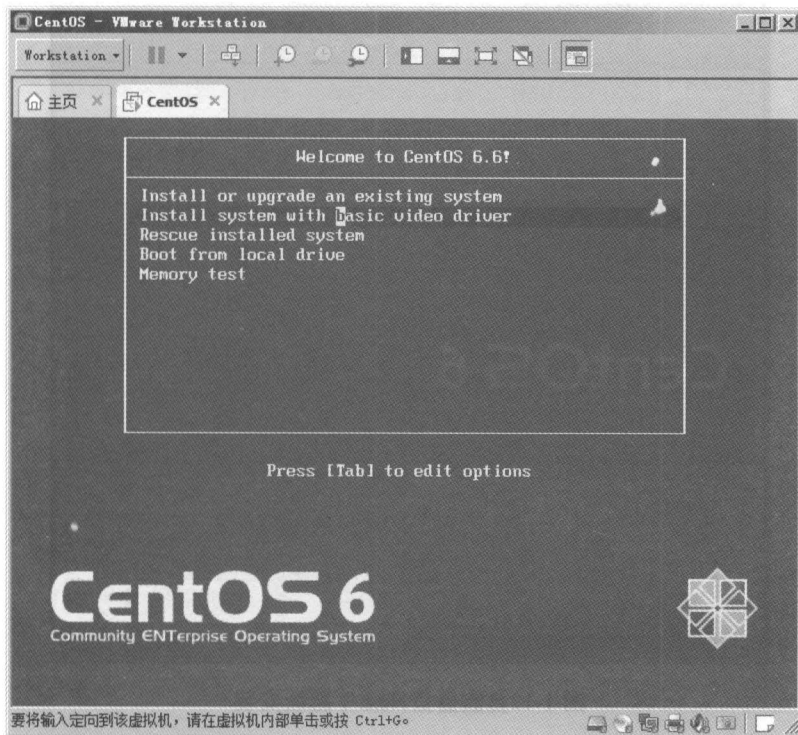


图 1-10 安装启动

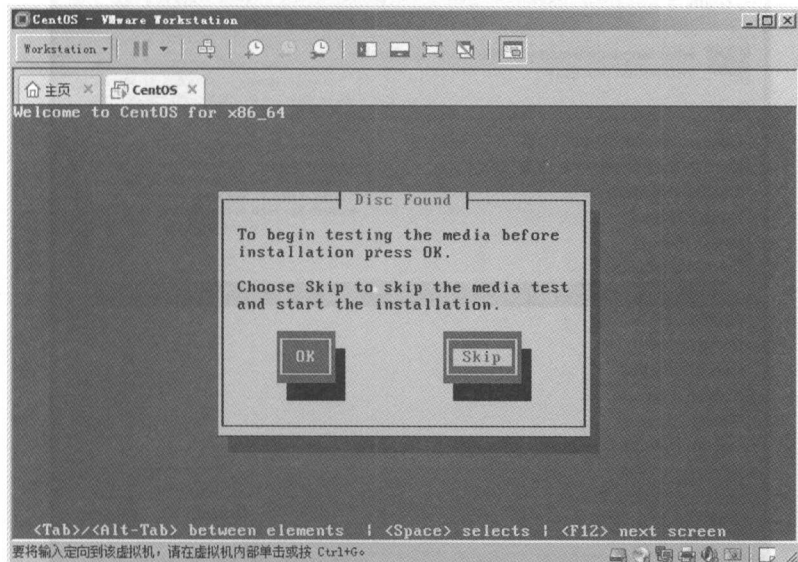


图 1-11 检测磁盘介质

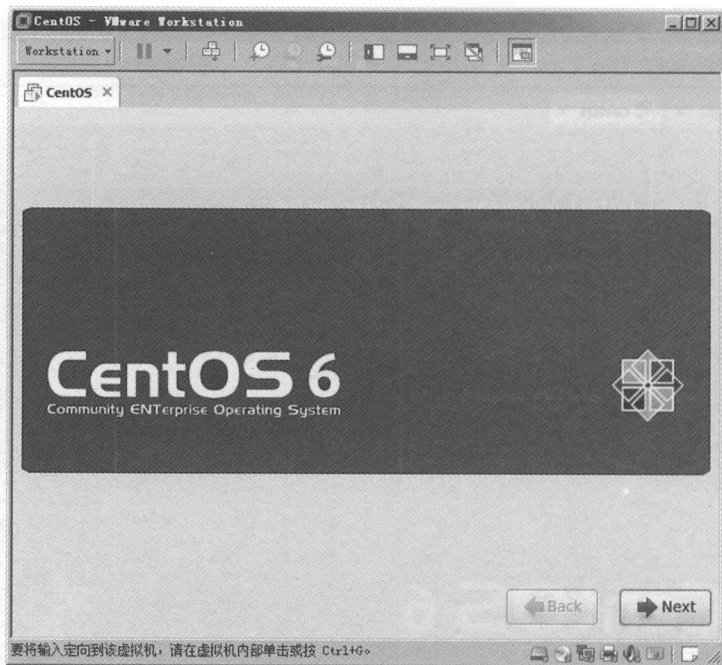


图 1-12 点击“Next”继续安装

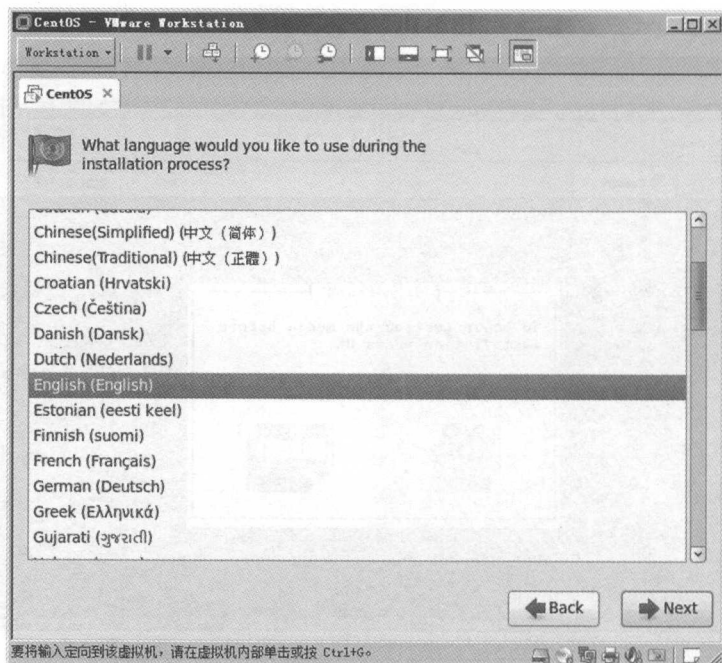


图 1-13 设置安装语言



图 1-14 设置键盘

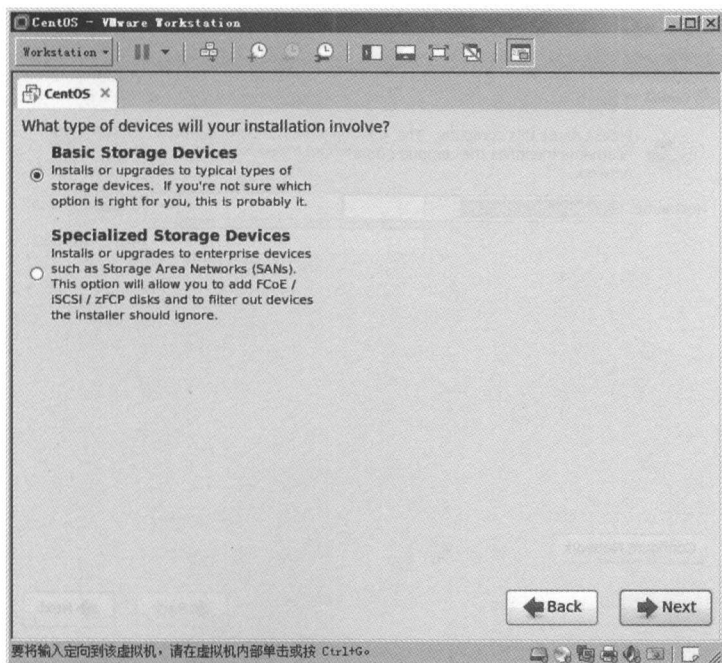


图 1-15 设置存储属性

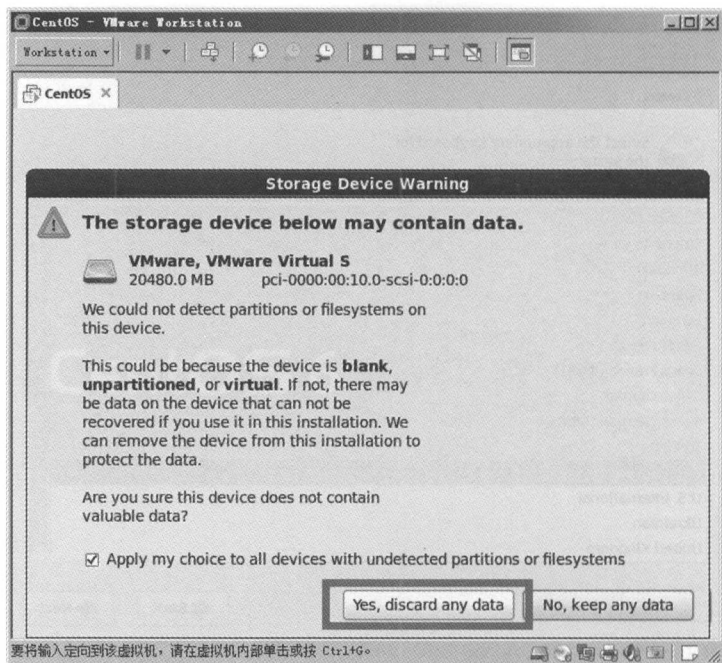


图 1-16 确认删除磁盘数据

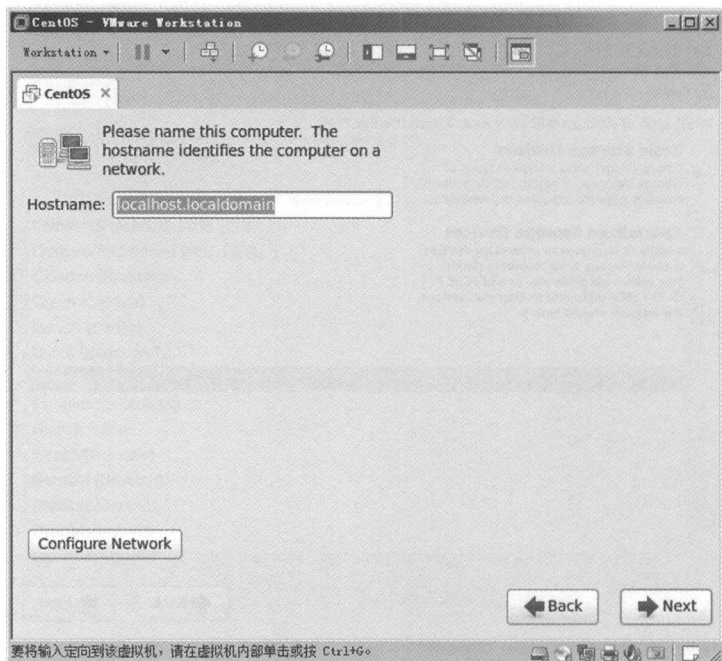


图 1-17 设置主机名

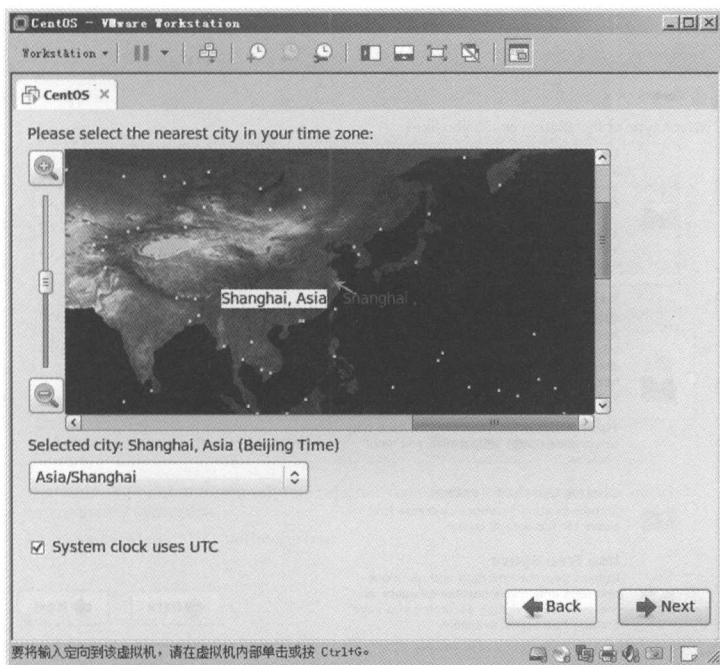


图 1-18 设置时区

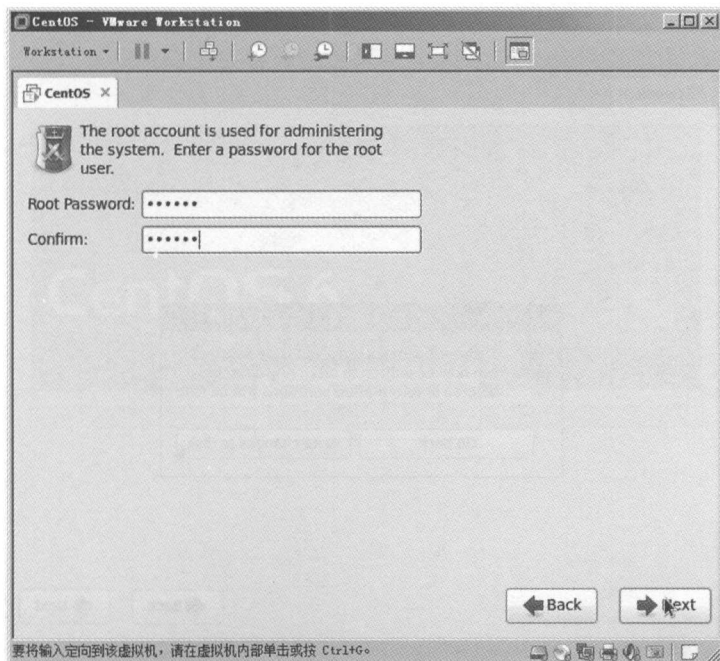


图 1-19 设置密码

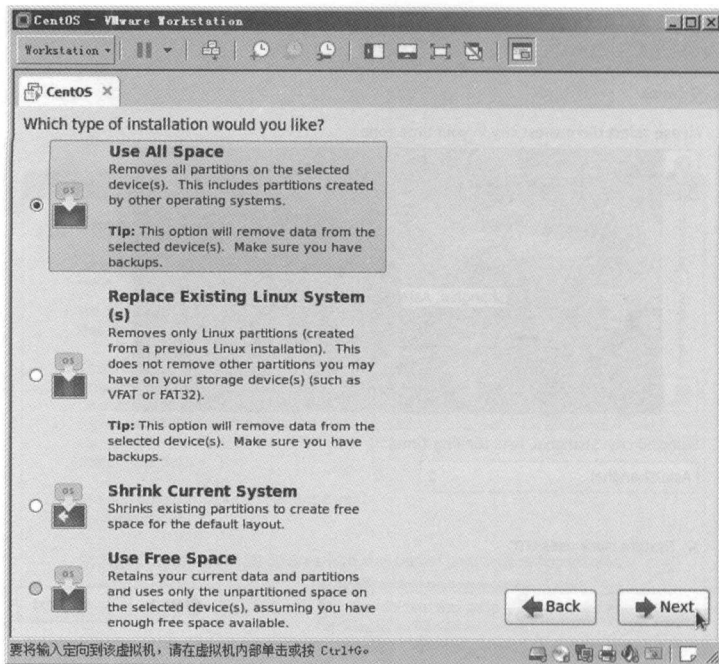


图 1-20 使用所有磁盘空间安装系统

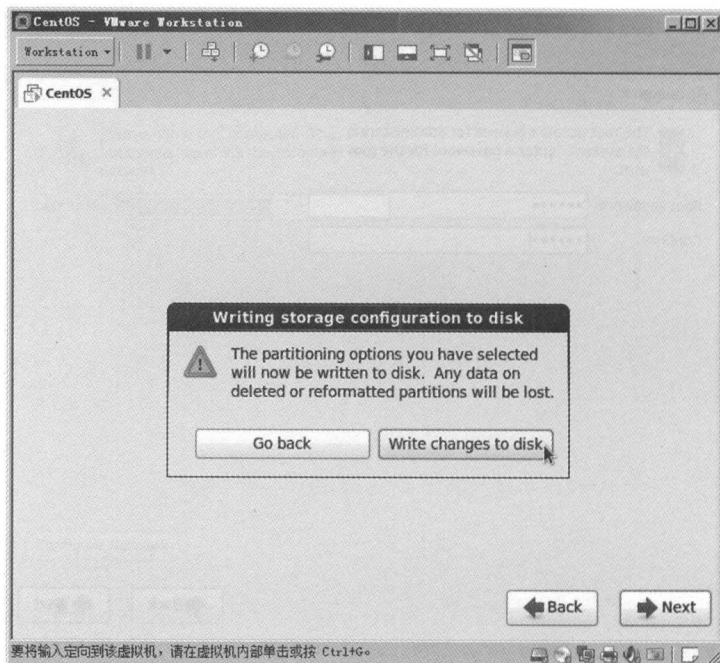


图 1-21 确认分区

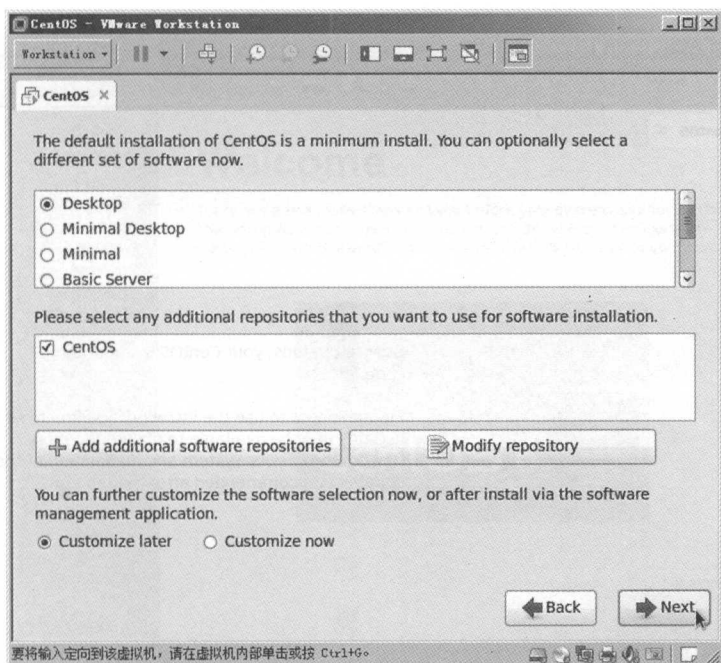


图 1-22 安装类型

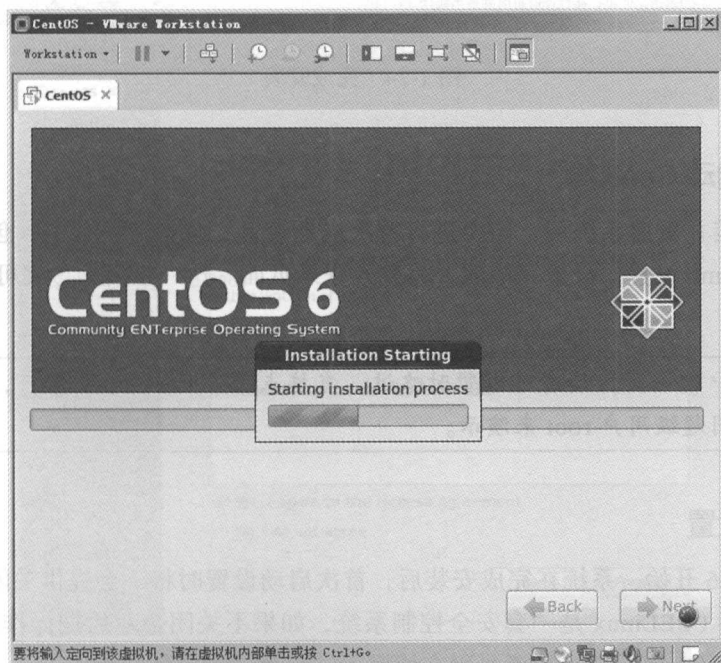


图 1-23 安装正式进行

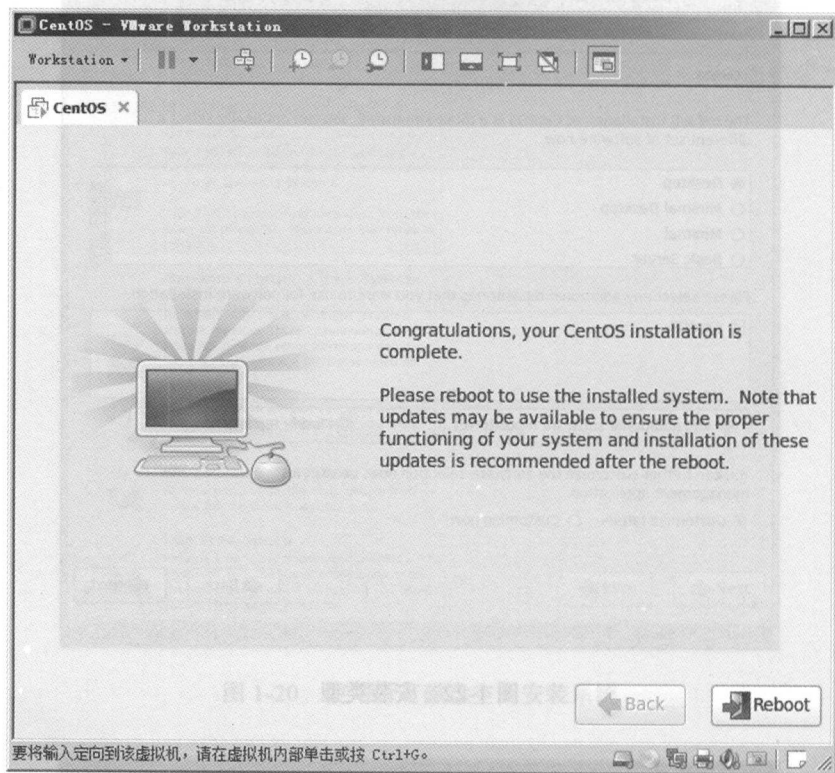


图 1-24 完成安装

1.1.2 首次启动 CentOS

在完成安装并重启系统后, 需要进行首次启动设置, 包括许可信息、创建用户、设置时间日期、Kdump 设置。设置完毕后, 将载入登录页面。这一系列的过程可参照图 1-25 至图 1-30 进行。



注意 创建用户这一页, 读者可以暂时略过, 直接点击“Forward”即可, 本书中所有操作将使用超级用户 root 来演示。

1.1.3 更多设置

从 CentOS 6 开始, 系统在完成安装后, 首次启动设置时将不会提供关闭防火墙、关闭 SELinux 的页面 (SELinux 是一套安全控制系统, 如果不关闭会对后期操作造成一些不便, 所以这里建议关闭) 等功能。读者可以在读完下一节后进行此处的操作。

图 1-31 和图 1-32 演示了如何通过图形页面配置系统防火墙。

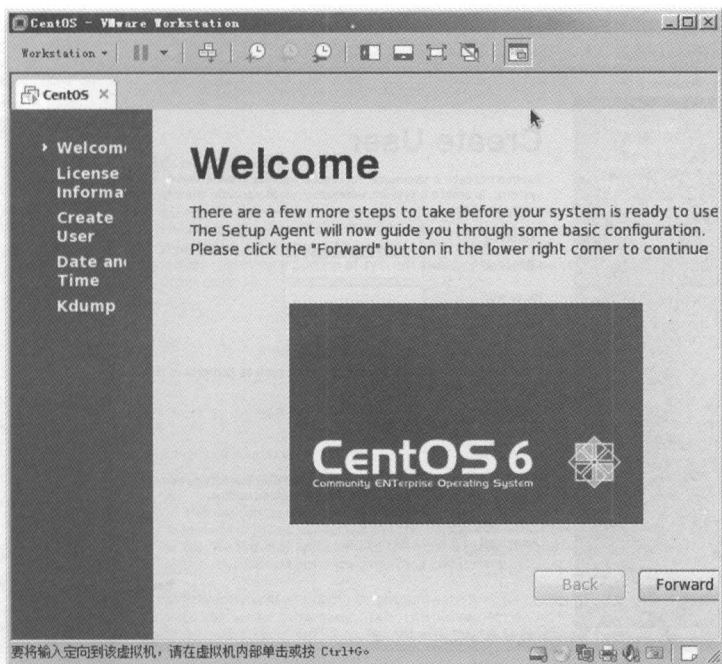


图 1-25 首次启动欢迎页面

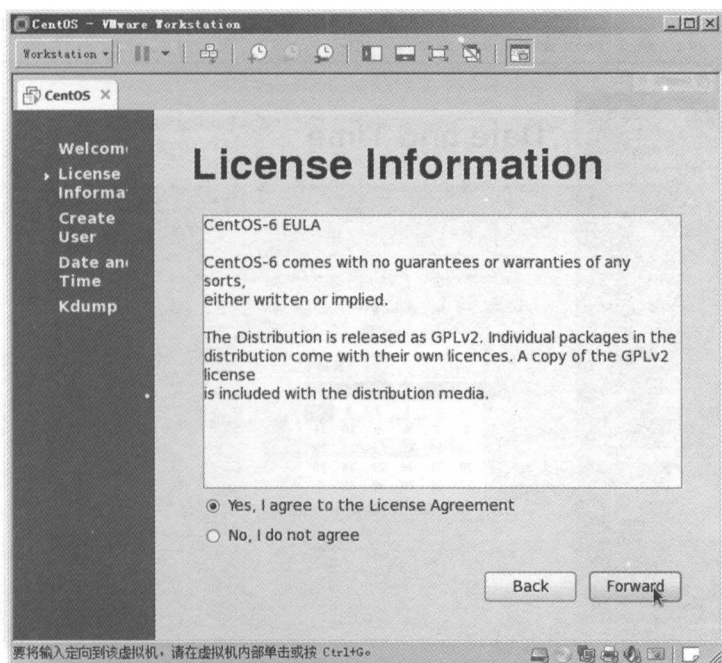


图 1-26 许可证

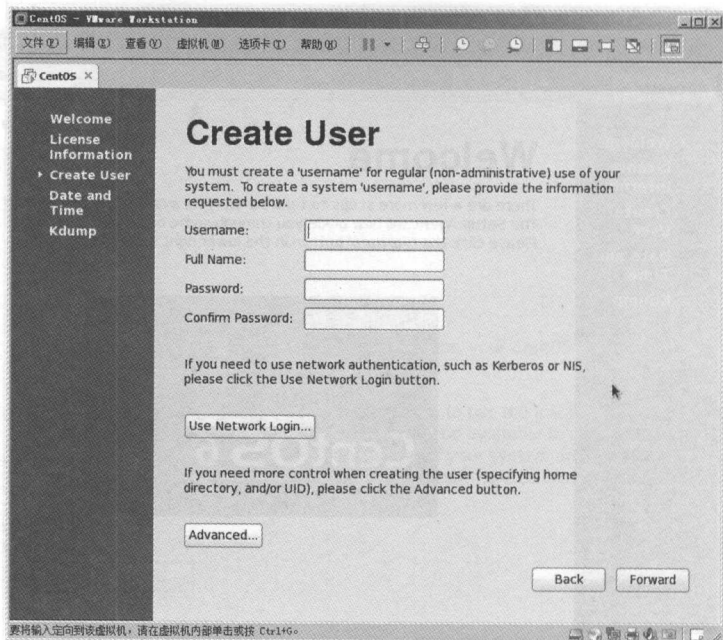


图 1-27 创建用户

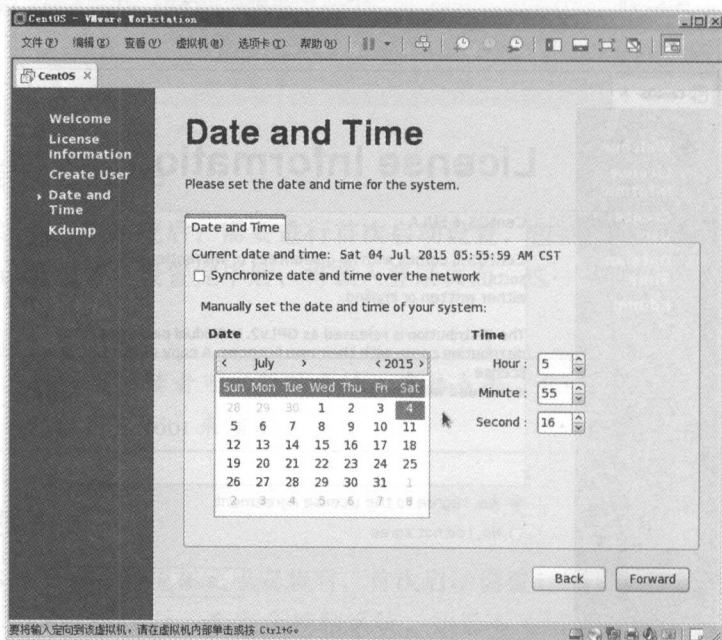


图 1-28 时间日期设置

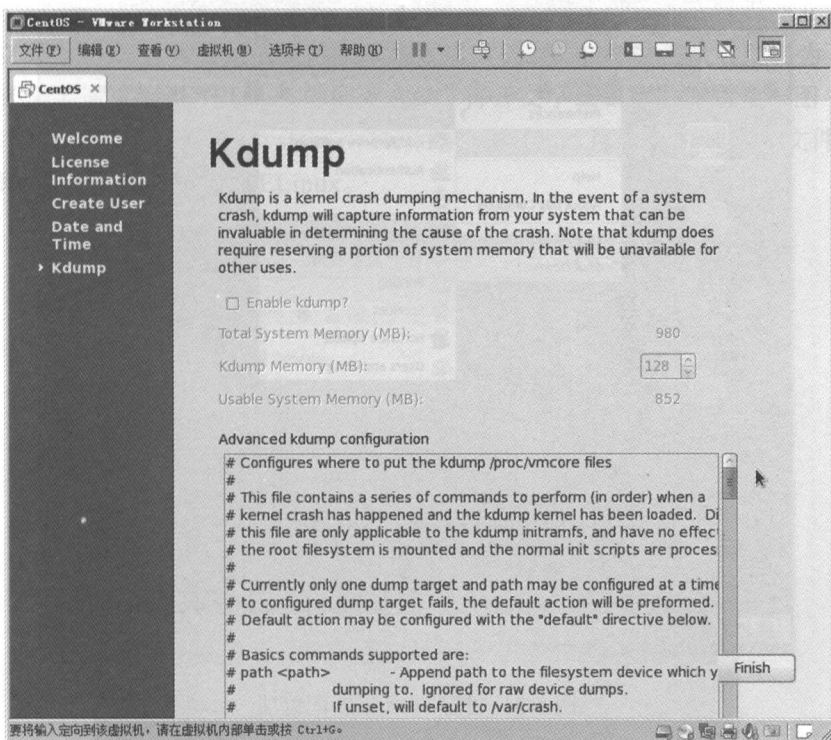


图 1-29 关闭 kdump 设置

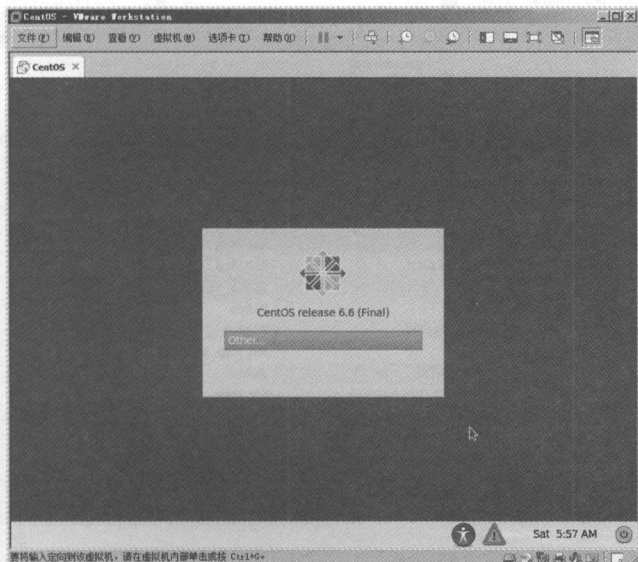


图 1-30 桌面载入

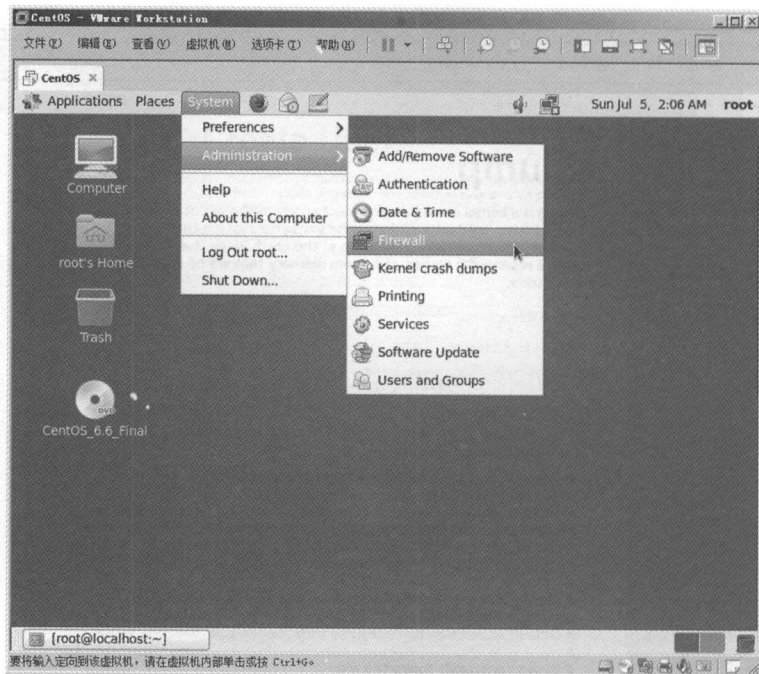


图 1-31 打开 Firewall 配置项

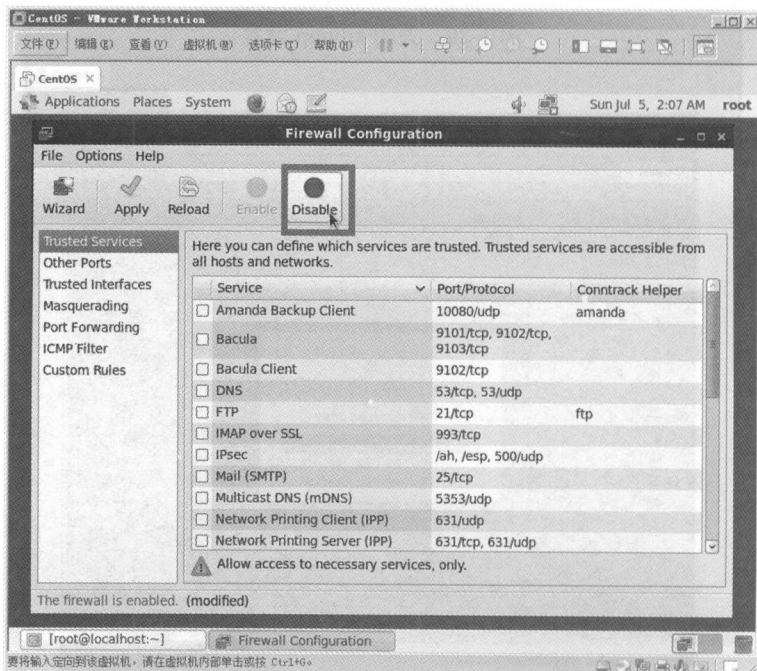


图 1-32 点击“Disable”并“Apply”

关闭防火墙后，再关闭 SELinux。可以在终端中使用命令“setenforce 0”立即关闭 SELinux（立即生效），这种方式的缺陷是系统重启后，SELinux 会再次启动，为了彻底关闭 SELinux，还需要通过编辑 SELinux 的配置文件（打开文字编辑器的方式参照图 1-33，文件具体路径参照图 1-34，即：File System → etc 目录→ selinux 目录下的 config 文件），图 1-33 到图 1-35 演示了如何彻底关闭 SELinux。

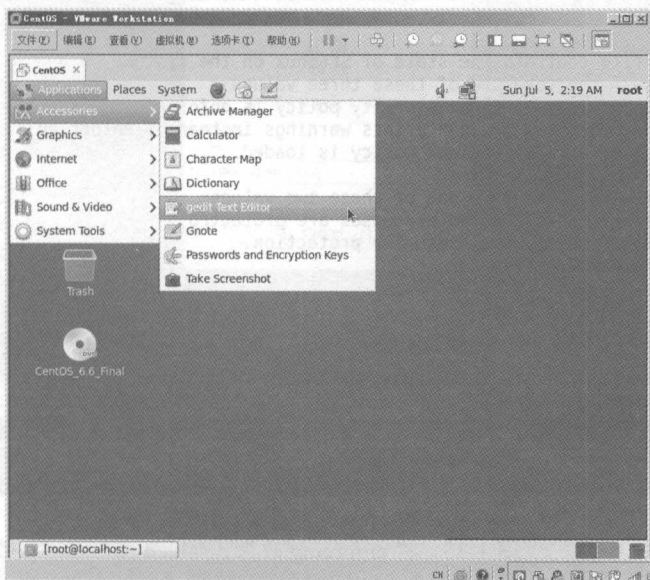


图 1-33 打开文字编辑器

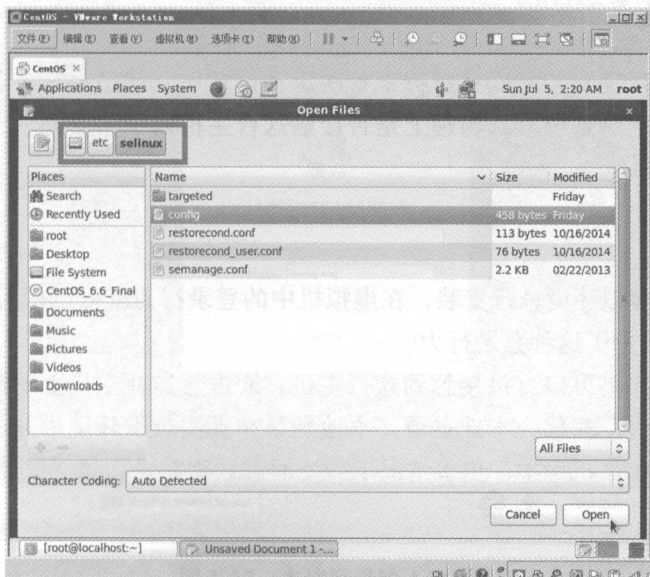


图 1-34 编辑 SELinux 配置文件

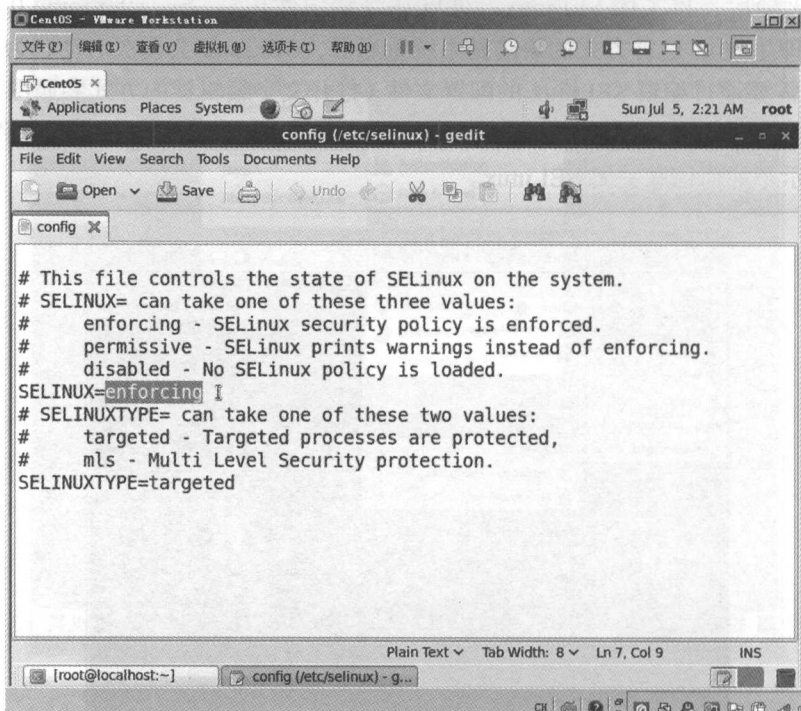


图 1-35 将 enforcing 改为 disabled

1.2 系统登录

如上节所示，系统安装完成并进行了首次启动设置后，就已经准备就绪了。我们所要做的就是了解如何登录系统。从物理上是否接触这台主机这个角度而言，系统登录分为本地登录和远程登录两种。

1.2.1 本地登录

如果读者使用虚拟环境进行安装，在虚拟机中的登录行为就可以认为是一个本地登录。图 1-36 和图 1-37 演示了这种登录行为。

更普遍的是，如果可以直接接触到这台主机，通过直连在主机上的键盘输入用户名和密码进行登录的行为，都称为本地登录。本地登录在真实运维环境中并不多见，因为主机往往是远程托管在 IDC 机房中，更多情况下需要通过远程登录。

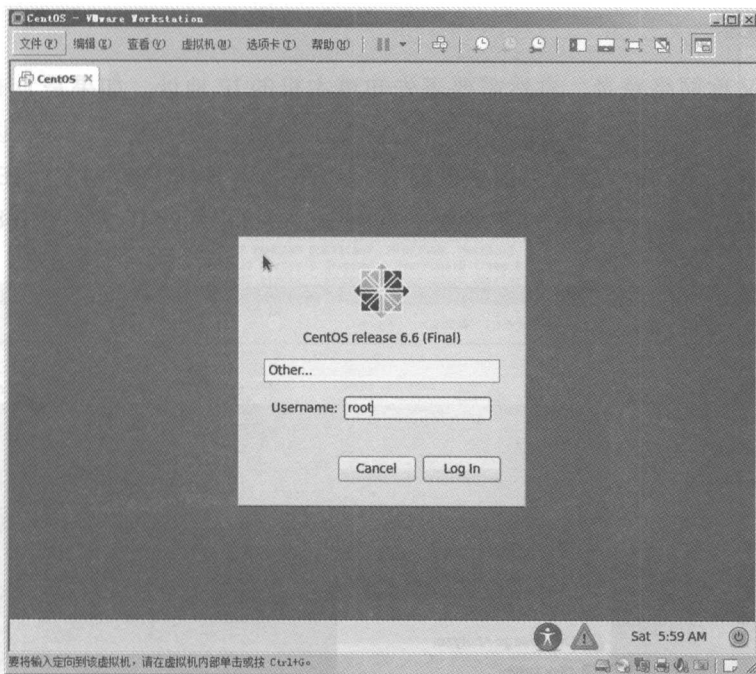


图 1-36 本地登录输入用户名

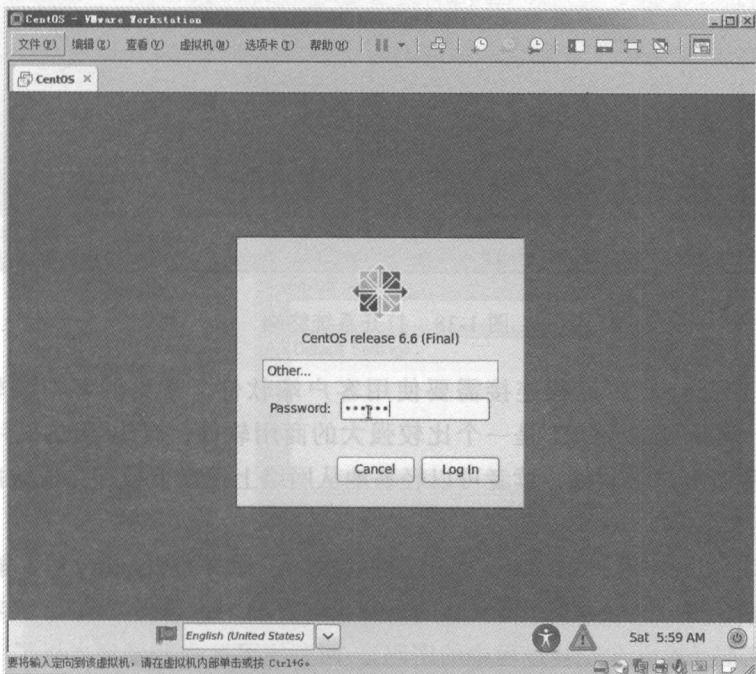


图 1-37 本地登录输入密码

1.2.2 远程登录

远程登录又称网络登录，自然需要事先知道主机的 IP 地址。如果是 DHCP 环境，只能通过本地登录查看主机的 IP 地址，因为 DHCP 动态给主机分配 IP 地址。读者可以通过 Terminal 使用 `ifconfig` 命令查看当前主机的 IP，如图 1-38 和图 1-39 所示。在实际运用中，更多的是给服务器配置一个固定的 IP 地址。本例中，主机得到的 IP 为 192.168.15.128。

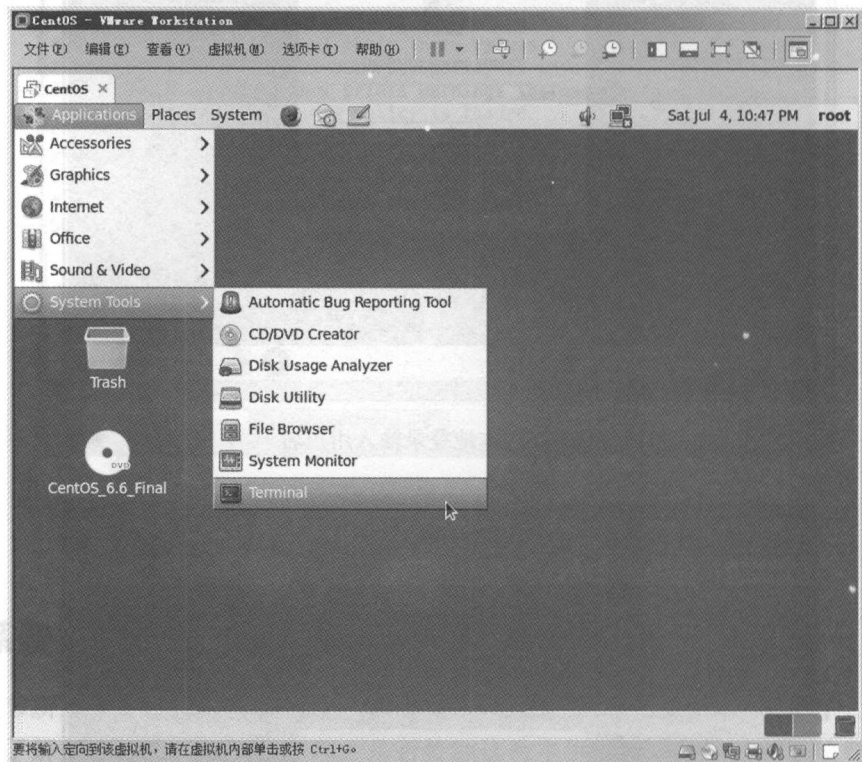


图 1-38 打开系统终端

得到主机 IP 后，要想远程连接需要使用客户端软件。常见的客户端软件有 `putty`、`SecureCRT` 等，其中 `SecureCRT` 是一个比较强大的商用软件，有 30 天的试用期限；`putty` 是一个小巧的免费客户端软件，读者可以轻易地从网络上搜索下载，并根据自己的喜好选择试用。

这里给读者演示如何试用 `putty` 远程连接主机。下载并打开 `putty` 后，输入想要连接的主机 IP，然后点击“Open”即可，在随后弹出的页面中，选择“是”，接着输入用户名 `root` 和正确的密码（输入在安装过程中的密码），回车便可远程登录到主机了（如图 1-40 至图 1-42 所示）。

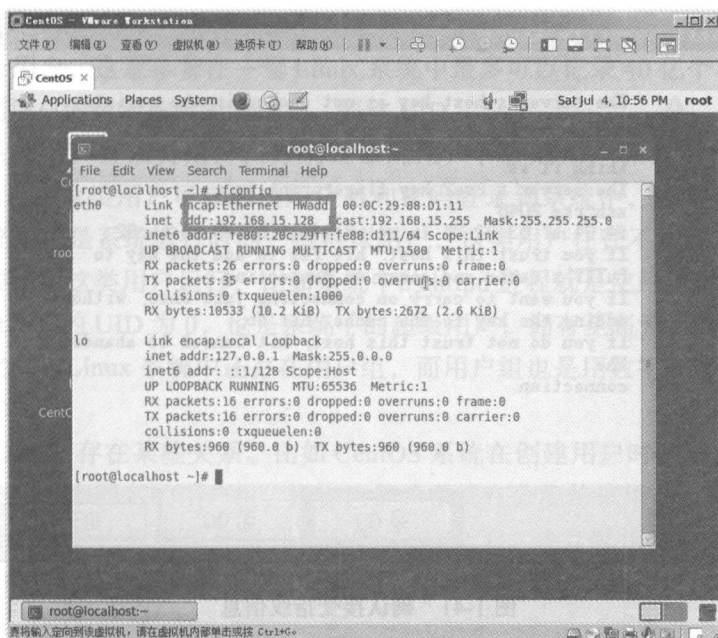


图 1-39 使用 ifconfig 命令查看 IP

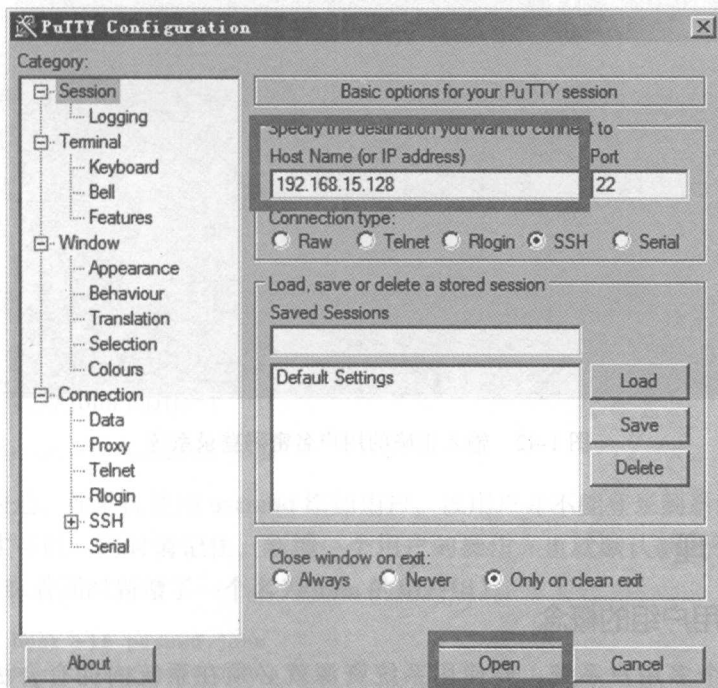


图 1-40 使用 putty 登录主机

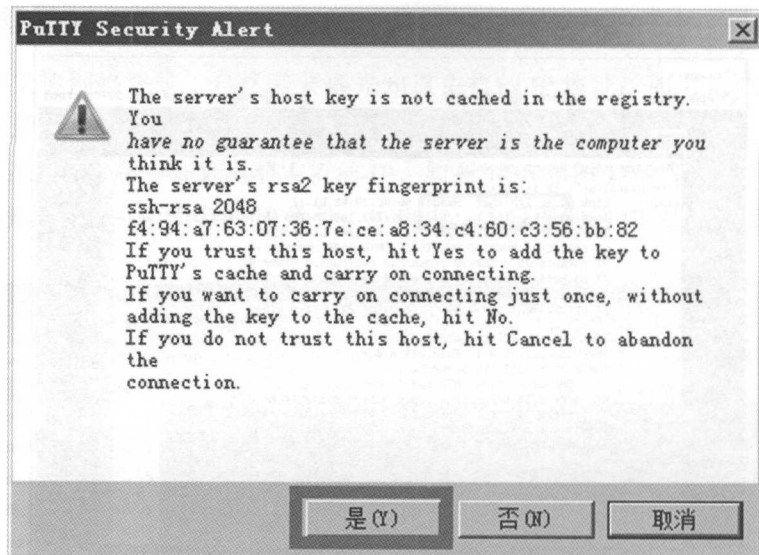


图 1-41 确认接受指纹信息

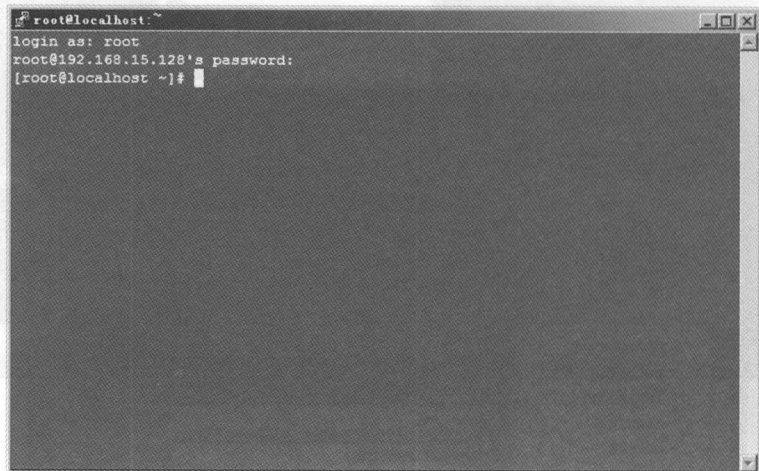


图 1-42 输入正确的用户名密码登录系统

1.3 用户管理

1.3.1 用户和用户组的概念

Linux 是一个多用户系统，要使用系统资源就必须在系统内拥有合法的用户账号，Linux 系统可以存在多个用户，但是需要使用唯一的用户名来区分不同的用户，同时所有非系统用户都需要设置密码才可以登录到系统。

和人类不同，Linux 系统只能使用数字来记录用户。在实现上，Linux 系统采用一个 32 位的整数来记录用户，这意味着在一套 Linux 系统中最多可以记录 40 亿个不同的用户。这个用来区分不同用户的数字被称为 user id，简称 UID。

在 Linux 系统中，有三类用户，分别是系统用户、普通用户和根用户。普通用户是 Linux 的真实用户，这类用户可以通过用户名和密码登录到系统中，通常普通用户的 UID 大于 500；系统用户是系统运行时的一些特殊用户，这类用户往往不能登录到系统中，但是一些进程需要使用这类用户运行，比如系统中的 httpd 进程就是使用用户 apache 运行的；根用户又叫 root，它的 UID 为 0，也是系统中的超级用户，拥有至高无上的权限。

除了用户之外，Linux 系统中还存在用户组，而用户组也是用数字来区分的，即 Group ID，简称为 GID。

UID 和 GID 之间存在某些关系。比如 CentOS 系统在创建用户时，系统会在创建这个用户的同时，创建一个同名的用户组。而在内部，系统在分配给该用户一个 UID 的同时会创建一个用户组（这个用户组也会得到一个唯一的 GID），并且默认情况下 UID 的值等于 GID，创建出来的这个用户默认属于这个用户组。用户组除了在创建用户时被创建，也可以独立创建出来。

上面的解释似乎有点晦涩，这里举个例子：在学校里，每个学生都会被分配到一个学号，这个学号一定是唯一的，所以才能区分不同的学生，我们可以拿学号类比一个 UID；同时，每个学生都可以自己创建自己的兴趣小组，这个兴趣小组的编号类比于系统中的 GID，为了保证唯一，创建的这个兴趣小组的编号的数值可以简单地等于 UID，这样可以保证 GID 也是唯一的。当然，默认情况下，一开始每个兴趣小组的成员都只有一个。当某个学生对其他某个兴趣小组感兴趣时，他可以随时加入其他的小组，这时该学生就属于两个组了，而他加入其他小组的个数越多，他从属于的组就越多，Linux 中也是一样，一个用户在创建后，至少属于一个组，而且后期随时可以加入、退出不同的组。

1.3.2 新增和删除用户

在 CentOS 中新增和删除用户可以分别使用 `useradd` 和 `userdel` 命令完成。比如现在想要添加一个用户名为 `john` 的用户：

```
[root@localhost ~]# useradd john
```

需要注意的是，如果仅使用 `useradd` 添加用户，该用户并不能登录到系统，必须给该用户设置密码后才可以。同时请记住，新增一个用户的操作，也就默认新增了一个同名的用户组（在这里意味着同时新增了一个名为 `john` 的用户组）。

```
[root@localhost ~]# passwd john
Changing password for user john.
New password:
Retype new password:
```

```
passwd: all authentication tokens updated successfully.
```

删除用户:

```
[root@localhost ~]# userdel john
```

在一个账号使用一段时间后, 该用户往往会在个人家目录中留下不少个人文件, 使用上面的命令删除用户, 这些文件还会得以保留。如果确认该用户的文件需要在删除用户时也一并彻底删除, 可使用以下命令完成:

```
[root@localhost ~]# userdel -r john
```

1.3.3 新增和删除用户组

也可以使用 `groupadd/groupdel` 单独创建 / 删除用户组。示例如下:

```
[root@localhost ~]# groupadd group1  
[root@localhost ~]# groupdel group1
```

1.3.4 用户切换

很多情况下需要切换用户, 比如原先使用了一个普通用户登录系统, 后来由于权限问题需要切换为 `root` 执行相关命令。或是需要从普通用户 1 切换为普通用户 2, 或是从 `root` 切换为普通用户等。切换用户的命令为 `su`。

`root` 由于拥有至高无上的权限, 所以, `root` 用户可以随时切换为任意的用户, 比如下面的例子中, `root` 用户切换为 `john`, 注意用户切换成功后, 命令提示行中的用户变为用户 `john` 了:

```
[root@localhost ~]# su - john  
[john@localhost ~]$
```

但是, 从普通用户切换至 `root`, 是必须要知道 `root` 的密码的, 下面的例子中第一次故意输入了一个错误的密码, 系统会拒绝这次用户切换; 第二次输入正确的密码后, 就可以正确切换为 `root` 了。

```
[john@localhost ~]$ su - root  
Password:  
su: incorrect password  
[john@localhost ~]$
```

```
[john@localhost ~]$ su - root  
Password:  
[root@localhost ~]#
```

最后, 从一个普通用户切换为另一个普通用户的操作, 也需要知道被切换的用户的密码, 原因应该很好理解。当然, 这里也存在一个很明显的问题: 用户 1 切换为用户 2 的前提是用户 1 必须知道用户 2 的密码, 这似乎给密码安全带来了一些问题。那么有没有方法

可以解决这个问题呢？答案是肯定的。感兴趣的读者可以搜索查看一下 `sudo` 命令。

1.4 文件系统

1.4.1 什么是文件系统

简单来说，文件系统是用来管理和组织文件的“方法”。Linux 支持多种不同的文件系统，常见的包括 `ext2`、`ext3`、`ext4`、`zfs`、`iso9600`、`msdos`、`ntfs`、`vfat`、`smbfs` 等，当然还可以通过加载模块的方式来支持更多的文件系统。虽然文件系统多种多样，但大部分 Linux 下的文件系统都有类似的结构，包括超级块、inode、数据块、目录块等。其中，超级块包括文件系统的总体信息，是文件系统的核心，所以磁盘中会有多个超级块，这样即使某一些超级块损坏了，文件系统依然可以使用。inode 存储所有与文件有关的数据，比如文件的权限和文件所指向的数据块等，也就是不包括文件真实内容和文件名。数据块是真实存储数据的部分，一个数据块默认的大小为 4KB。目录块包括文件名和文件在目录中的位置，以及 inode 的信息。

1.4.2 常见的文件系统

1. ext2 文件系统

Linux 最早引入的文件系统类型是 minix，由于其存在一定的局限性，比如文件名最长仅支持 14 个字符，文件最大为 64MB 等因素，后来被 `ext2` (The Second Extended File System) 文件系统所取代，该系统有着极好的存储性能，所以曾一度成为 Linux 中的标准文件系统。和很多文件系统一样，`ext2` 文件系统也是采取将文件数据存放到数据块中的方式来存储数据的，这些数据块的大小可以在创建文件系统的时候指定，对于存放的每个文件和目录，都会有一个 inode 指定，文件系统中所有的 inode 都是使用 inode 表来进行记录的，一定数量的块就会组成一个块组。在 `ext2` 文件系统中，整个分区的文件系统信息都被存放在超级块中，考虑到超级块的重要性，在每个块组的开头中都有相同的备份。

但是 `ext2` 文件系统的弱点也是很明显的，它不支持日志功能，这很容易造成在一些极端场景中丢失数据，这个天然的弱点导致 `ext2` 文件系统无法在关键应用中使用，目前已经很少有企业使用 `ext2` 文件系统了。

2. ext3 文件系统

为了弥补 `ext2` 文件系统的不足，有日志功能的 `ext3` 文件系统应运而生了。它直接从 `ext2` 文件系统发展而来，所以完全兼容 `ext2` 文件系统，而且支持非常简单地从 `ext2` 转换为 `ext3` (只需要两条命令)，这种特性让也更多的老用户转而使用 `ext3` 文件系统。

那么为什么需要日志文件系统呢？因为日志文件系统使用了“两阶段提交”的方式来维护待处理的事物。例如在写入数据之前，文件系统会先在日志中写入，然后再开始真实

地写数据，写完数据后则会将之前写入日志中的内容删除。这样一来，如果遇到问题需要检查文件系统或对 ext3 文件系统进行修复时，只需要检查日志即可。而 ext2 修复文件系统时，需要遍历整个文件系统来检查文件的一致性信息，因此 ext3 节省了大量修复文件系统所需的时间。不过，由于增加了日志功能，在存取数据时 ext3 文件系统看起来要比 ext2 多一次写入操作，但是 ext3 对写操作做了优化，所以其性能并不比 ext2 低。

3. ext4 文件系统

ext4 文件系统从 2.6.19 内核开始引入，从 CentOS 6 开始，ext4 也已经成为默认的文件系统。和 ext2 到 ext3 的升级一样，从 ext3 到 ext4 也是可以在线迁移的，和 ext3 相比，ext4 支持 1EB 的文件系统，以及 16TB 的文件，同时支持无数量限制的子目录。

1.4.3 磁盘分区和创建文件系统

磁盘使用前需对其进行分割，这种动作被形象地称为分区。磁盘的分区分为两类，即主分区和扩展分区。受限于磁盘的分区表大小（MBR 大小为 512 字节，其中分区表占 64 字节），一块磁盘最多只能创建 4 个主分区，为了能支持更多分区，可以使用扩展分区（扩展分区中可以划分更多逻辑分区），但是即便这样，分区还是要受主分区 + 扩展分区最多不能超过 4 个的限制。磁盘在完成分区后，需要进行创建文件系统的操作，最后将该分区挂载到系统中的某个挂载点才可以使用。

下面继续使用 VMware 虚拟机演示 fdisk 的使用，首先会在虚拟机设置中添加一块磁盘，方式如图 1-43 至图 1-48 所示，完成后启动虚拟机。

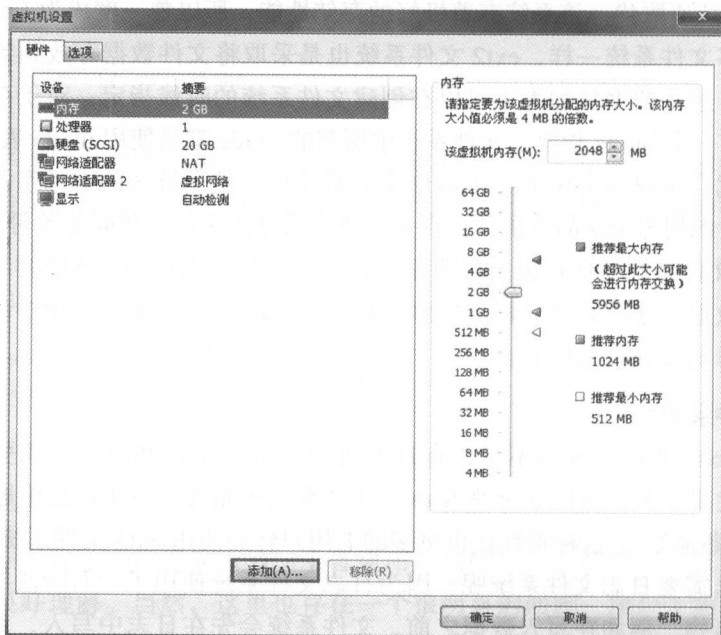


图 1-43 选择“添加”，为虚拟机增加设备

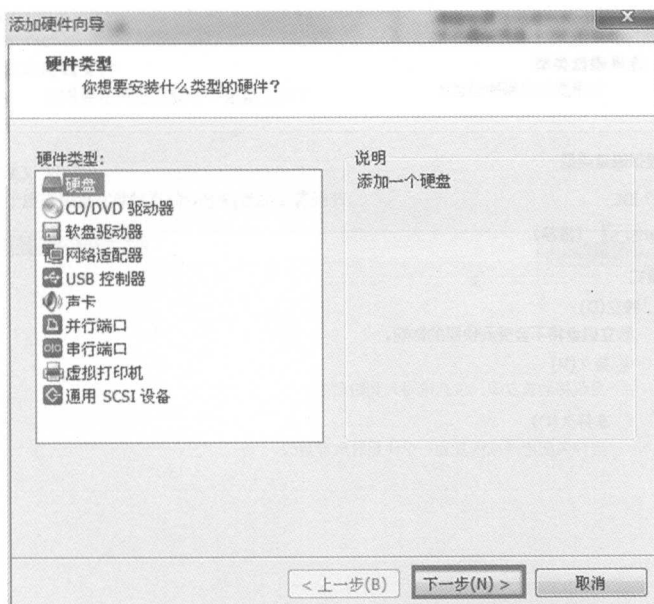


图 1-44 选择“硬盘”

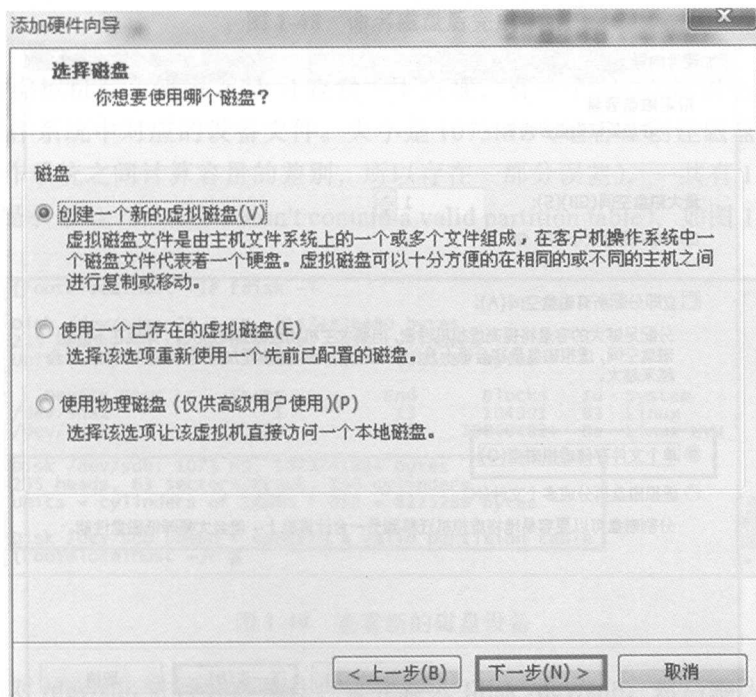


图 1-45 选择“创建一个新的虚拟磁盘”

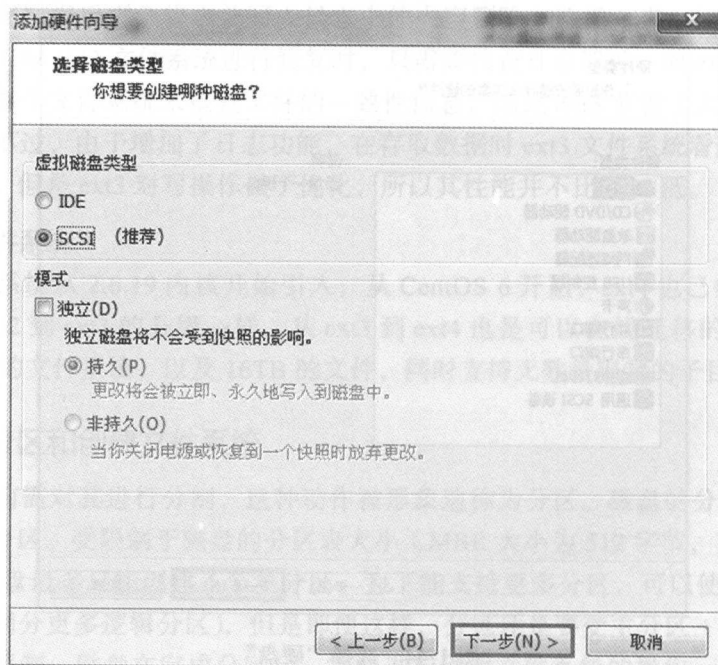


图 1-46 选择“SCSI”

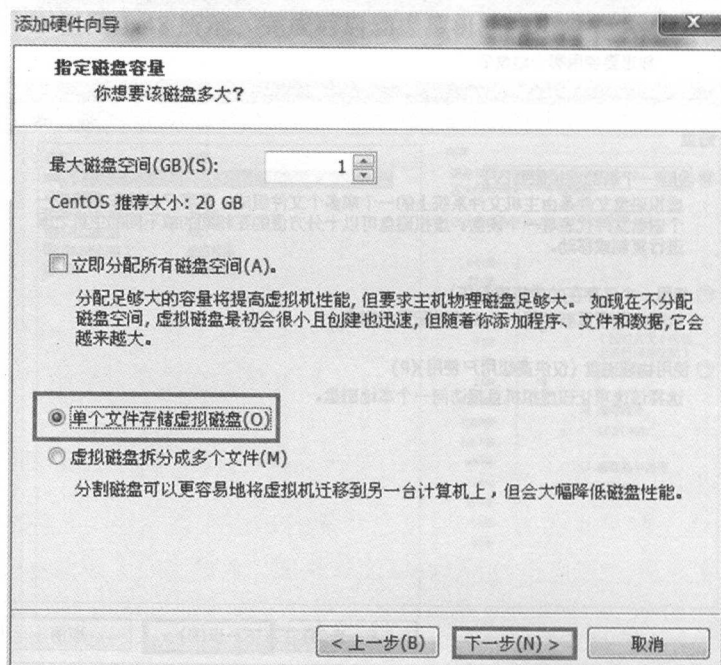


图 1-47 选择“单个文件存储虚拟磁盘”

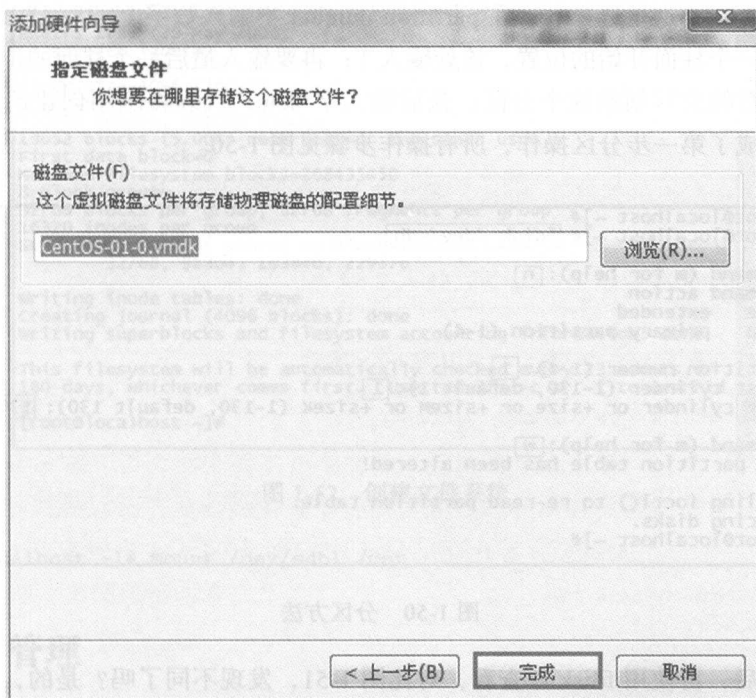


图 1-48 命名磁盘后完成

重新启动虚拟机后，使用 `fdisk -l` 查看一下发现，有一个 `/dev/sdb` 设备，这就是新添加的磁盘在操作系统中对应的设备文件。大小是 1073MB（笔者在创建磁盘时给的大小是 1GB，由于操作系统之间计算容量的差别，所以存在一部分误差），一共有 130 个柱面，而且没有分区（提示 `Disk /dev/sdb doesn't contain a valid partition table`），如图 1-49 所示。

```
[root@localhost ~]# fdisk -l
Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1 *           1           13       104391    83  Linux
/dev/sda2             14        2610     20860402+   8e  Linux LVM

Disk /dev/sdb: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

Disk /dev/sdb doesn't contain a valid partition table
[root@localhost ~]#
```

图 1-49 查看新的磁盘设备

下面开始对 `/dev/sdb` 进行分区操作，首先输入 `fdisk /dev/sdb`，然后输入字母 `n`，这个字母代表 `new`，也就是新建分区；然后系统会提示是创建扩展分区（`extended`）还是主分区

(primary partition), 这里选择 p; 在 partition number 中输入数字 1, 代表这是第一个分区; 下面要输入第一个柱面开始的位置, 该处输入 1; 再要输入最后一个柱面的位置, 这里输入 130 表示将所有的空间划给这个分区; 最后输入字母 w, 代表将刚刚创建的分区写入分区表。这样就完成了第一步分区操作, 所有操作步骤见图 1-50。

```
[root@localhost ~]#
[root@localhost ~]# fdisk /dev/sdb

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-130, default 1): 1
Last cylinder or +size or +sizeM or +sizeK (1-130, default 130): 130

Command (m for help): w
The partition table has been altered!

calling ioctl() to re-read partition table.
Syncing disks.
[root@localhost ~]#
```

图 1-50 分区方法

分区完成后, 再使用 fdisk -l 查看, 对比图 1-51, 发现不同了吗? 是的, 这里显示出一个设备, 叫做 /dev/sdb1, 这就是下一步需要创建文件系统的设备。

```
[root@localhost ~]# fdisk -l

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks    Id  System
/dev/sda1  *           1           13        104391    83  Linux
/dev/sda2             14        2610    20860402+   8e  Linux LVM

Disk /dev/sdb: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks    Id  System
/dev/sdb1             1          130    1044193+    83  Linux

[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]#
```

图 1-51 确认磁盘分区成功

要在刚刚创建的分区中格式化文件系统, 这里使用 ext3 文件系统作为演示, 可以使用命令 “mkfs -t ext3 /dev/sdb1” 来完成, 或是简单地将此命令写成 “mkfs.ext3 /dev/sdb1”, 这两个命令是一样的, 如图 1-52 所示。

成功创建文件系统后才可以将磁盘挂载到挂载点, 假设这里需要将 /dev/sdb1 挂载到 /mnt, 可以使用以下命令:

```
[root@localhost ~]# mkfs.ext3 /dev/sdb1
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
130560 inodes, 261048 blocks
13052 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
16320 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376

writing inode tables: done
Creating journal (4096 blocks): done
writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 23 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to overrid
e.
[root@localhost ~]#
```

图 1-52 创建文件系统

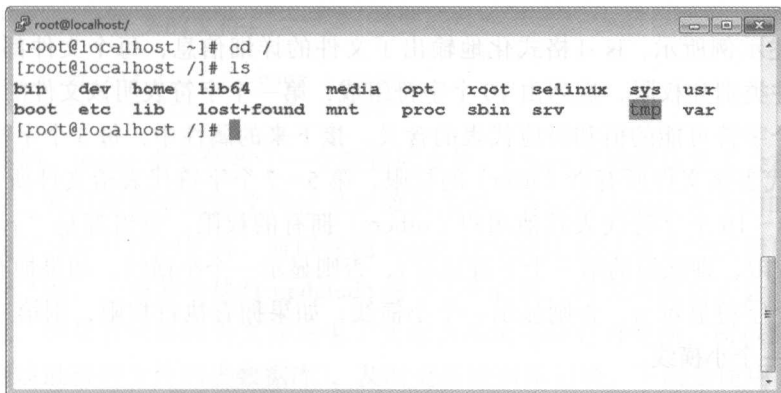
```
[root@localhost ~]# mount /dev/sdb1 /mnt
```

1.5 文件管理

1.5.1 文件和目录简介

Linux 使用树状的目录结构组织文件，简单来说就是在一个目录中放置子目录和文件，子目录中可以继续放置子目录和文件，以此类推，形似一棵树的分支（如图 1-53 所示）。Linux 的这种文件结构的起始点为“根目录”，就是“/”，是一切文件的起点。FHS（文件系统层次标准）定义了在根目录下的主要目录和每个目录内应该放置的文件。

请注意在 Linux 中，“文件”是一种很宽泛的概念，一切皆文件。所以不管是目录还是设备，都是一种文件，或者说，只要是在系统中可以看到的都是文件。



```
root@localhost/
[root@localhost ~]# cd /
[root@localhost /]# ls
bin  dev  home  lib64      media  opt   root  selinux  sys  usr
boot  etc  lib  lost+found  mnt    proc  sbin  srv      tmp  var
[root@localhost /]#
```

图 1-53 查看根目录中的文件

对于系统中任何具体的文件来说，都一定可以通过绝对路径找到。比如系统引导文件 `grub.conf`。

```
[root@localhost ~]# ls /boot/grub/grub.conf
/boot/grub/grub.conf
```

但是假设现在所在的目录是 `/boot`，那么该文件就可以使用相对路径找到。

```
[root@localhost ~]# cd /boot
[root@localhost boot]# ls ./grub/grub.conf
./grub/grub.conf
```

这里的点 (.) 代表当前目录，很多时候可以省略。

```
[root@localhost boot]# ls grub/grub.conf
grub/grub.conf
```

想要回到当前目录的上层目录，使用两个点 (..)，代表上层目录，也是一个相对路径的写法。

```
[root@localhost boot]# cd ..
[root@localhost /]#
```

1.5.2 文件和目录权限

使用 `ls` 命令，结合 “`-l`” 参数查看文件的权限，结合 “`-ld`” 参数查看目录的权限，查看 `/root` 目录以及查看 `/root` 目录下文件的权限如下：

```
[root@localhost ~]# ls -l
total 20
-rw-----. 1 root root 1122 Jul  6 04:57 anaconda-ks.cfg
-rw-r--r--. 1 root root 9562 Jul  6 04:57 install.log
-rw-r--r--. 1 root root 3161 Jul  6 04:56 install.log.syslog
[root@localhost ~]# ll -d /root
dr-xr-x---. 2 root root 4096 Jul  6 05:00 /root
```

正如上述示例所示，`ls -l` 格式化地输出了文件的详细信息，每个文件都有 7 列输出。第一列是文件类别和权限，这列由 10 个字符组成，第一个字符表明该文件的类型。表 1-1 列出了第一个字符可能的值和对应代表的含义。接下来的属性中，每 3 个字符为一组，第 2~4 个字符代表该文件所有者 (user) 的权限，第 5~7 个字符代表给文件所有组 (group) 的权限，第 8~10 个字符代表其他用户 (others) 拥有的权限。每组都是 “`rwX`” 的组合，如果拥有读权限，则该组的第一个字符显示 `r`，否则显示一个小横线；如果拥有写权限，则该组的第二个字符显示 `w`，否则显示一个小横线；如果拥有执行权限，则第三个字符显示 `x`，否则显示一个小横线。

表 1-1 文件权限首字符含义

第一个字符可能的值	含 义
d	目录
-	普通文件
l	链接文件
b	块文件
c	字符文件
s	socket 文件

1.5.3 文件查找

操作系统中有成千上万的文件散落在文件系统的各个角落中，还有不同用户创建的各种文件，随着系统的运行，文件数会越来越多，要想记住所有文件的位置是不可能的，但我们可以通过一些查找命令来进行。最常用的命令有 `find` 和 `locate`。

其中 `find` 命令的使用格式为：

```
find PATH -name FILENAME
```

假设需要在系统中找到一个名为 `httpd.conf` 的文件，可以这么写：

```
[root@localhost ~]# find / -name httpd.conf
```

这条命令的意思是，从根目录开始寻找名字为 `httpd.conf` 的文件。由于是从根开始，`find` 命令会遍历 / 下的所有文件，然后打印出找到的结果。如果读者有经验，大概知道这个文件可能存在于 `/etc` 下，因为看起来这是一个配置文件，这时便可以优化一下查找语句，这样耗时会更少，命令如下所示：

```
[root@localhost ~]# find /etc -name httpd.conf
```

可以使用星号通配符来模糊匹配要查找的文件名，比如想找出系统中所有以 `.conf` 结尾的文件，或是以 `httpd` 开头的文件：

```
[root@localhost ~]# find / -name *.conf
[root@localhost ~]# find / -name httpd*
```

和 `find` 不同，`locate` 命令依赖于一个数据库文件，Linux 系统默认每天会检索一下系统中的所有文件，然后将检索到的文件记录到数据库中，在运行 `locate` 命令的时候可直接到数据库中查找记录并打印到屏幕，所以使用 `locate` 命令要比 `find` 命令的反馈更为迅速。在执行这个命令之前，一般需要执行 `updatedb` 命令（非必须，因为系统每天会自动检索并更新数据库信息，但是有时候因为文件发生了变化而系统还没有再更新，所以需要主动运行该命令，以创建最新的文件列表数据库），及时更新数据库记录，下面是使用 `locate` 命令查找 `httpd.conf` 文件的方法：

```
[root@localhost ~]# updatedb
[root@localhost ~]# locate httpd.conf
/etc/httpd/conf/httpd.conf
```

locate 命令依赖于其用于记录文件的数据库，该数据库的更新需要使用 updatedb。当然，系统每天也会自动运行一次，必要的时候可主动地手动更新。

对一些二进制的命令文件，可以通过 which 命令找到。which 用于从系统的 PATH 变量所定义的目录中查找可执行文件的绝对路径。例如想查找 passwd 命令在系统中的绝对路径，可使用如下方法：

```
[root@localhost ~]# which passwd
/usr/bin/passwd
```

1.5.4 文件压缩和打包

gzip/gunzip 是用来压缩和解压单个文件的工具，使用方法比较简单，例如在 /root 目录下压缩 install.log 文件，压缩后产生的文件是 install.log.gz 文件，然后再使用 gunzip 文件将其解压缩：

```
[root@localhost ~]# gzip install.log
[root@localhost ~]# ls install.log.gz
install.log.gz
[root@localhost ~]# gunzip install.log.gz
```

tar 不但可以打包文件，还可以将整个目录中的全部文件整合成一个包，整合包的同时还能使用 gzip 的功能进行压缩，例如把整个 /boot 目录整合并压缩成一个文件。一般来说，对于整合后的包，业内习惯使用 .tar 作为其后缀名，使用 gzip 压缩后的文件则使用 .gz 作为其后缀名。因为 tar 有同时整合和压缩的功能，所以可使用 .tar.gz 作为后缀名，或者简写为 .tgz，下面的命令将 /boot 目录整合压缩成了 boot.tgz 文件：

```
[root@localhost ~]# tar -zcvf boot.tgz /boot
```

这里 -z 的含义是使用 gzip 压缩，-c 是创建压缩文件（create），-v 是显示当前被压缩的文件，-f 是指使用文件名，也就是这里的 boot.tgz 文件。解压命令如下：

```
tar -zxvf boot.tgz
```

上面的命令会直接将 boot.tgz 在当前目录中解压成 boot 目录，-z 是解压的意思。如需指定压缩后目录存放的位置，需要再使用 -C 参数，比如说将 boot 目录解压到 /tmp 目录中：

```
tar -zxvf boot.tgz -C /tmp
```

使用 bzip2 压缩文件时，默认会产生以 .bz2 扩展名结尾的文件，这里使用 -z 参数进行压缩，使用 -d 参数进行解压缩：

```
[root@localhost ~]# bzip2 install.log
[root@localhost ~]# ls -l install.log.bz2
```

```
-rw-r--r-- 1 root root 3588 Dec 10 03:08 install.log.bz2
[root@localhost ~]# bzip2 -d install.log.bz2
```

1.6 网络管理

网络才能让 Linux 发挥最大的功能，所以了解 Linux 的网络配置也是必备技能，这就需要记住一些常见的配置命令和相关的配置文件。

1.6.1 网络配置管理

使用 `ifconfig` 命令可以看到当前系统所有活动网卡的状态（如图 1-54 所示）。其中 `eth0` 表示的是一块网卡，如果有多个网卡，系统会自动往后标注 `eth1`、`eth2`，以此类推。

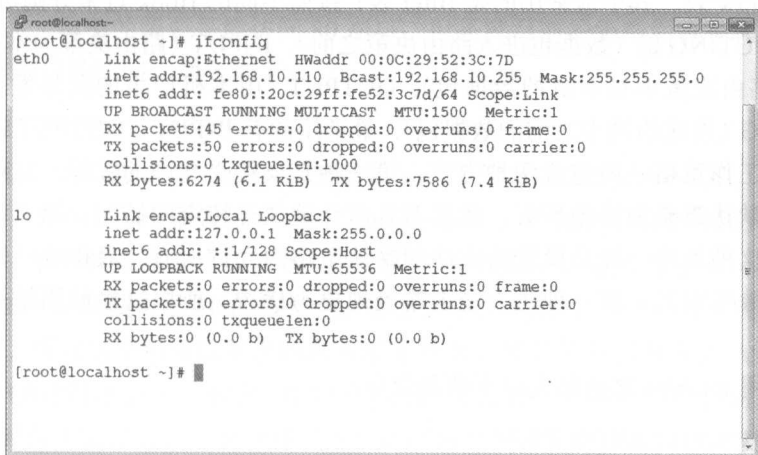


图 1-54 查看活动网卡

前面的安装过程中，并没有对网络进行过任何配置，所以现在主机得到的 IP 是采用了默认的 DHCP 协议得到的，读者使用这个命令看到的 IP 和图 1-54 所示的可能不一致。也可以使用 `ifconfig` 给主机主动配置 IP 地址：

```
[root@localhost ~]# ifconfig eth0 192.168.10.130 netmask 255.255.255.0
#该命令可以简写为：
#[root@localhost ~]# ifconfig eth0 192.168.10.130/24
```

`ifconfig` 命令是及时生效的，所以如果是通过远程连接的主机，现在要做 IP 的修改，一旦回车当前的连接就中断了，需要重新登录到新的 IP 才行。而且这个命令并不会帮助用户把 IP 记录到任何配置文件中，也就是说，一旦主机重启，主机依然会从 DHCP 配置 IP，所以如果想使用一个固定的 IP，则需要手动将该 IP 写到配置文件中。

CentOS 的网络配置相关文件集中存放在 `/etc/sysconfig/network-scripts` 目录中，而 `eth0`

的配置文件就是该目录下的 ifcfg-eth0，如果要将该主机的 IP 固定配置为 192.168.10.130，则可以按照如下配置：

```
DEVICE=eth0
BOOTPROTO=static
ONBOOT=yes
IPADDR=192.168.10.130
NETMASK=255.255.255.0
```

1.6.2 Linux 防火墙

iptables 是 Linux 下功能强大的防火墙工具，由于它集成于 Linux 内核，所以效率极高。该工具在系统安装的过程中会默认安装。iptables 按照对数据包的操作分为 4 个表，这 4 个表分别是 filter 表（用于一般的过滤）、nat 表（地址或端口映射）、mangle 表（对特定数据包的修改）、raw 表，其中最常用的是 filter 表；按照不同的 Hook 点来分则可分为 5 个链，分别是 PREROUTING 链（数据包进入路由决策之前）、INPUT（路由决策为本机的数据包）、FORWARD（路由决策不是本机的数据包）、OUTPUT（由本机产生的向外发送的数据包）、POSTROUTING（发送给网卡之前的数据包），最常用的有 INPUT、OUTPUT 链。

防火墙的工作策略一般包含两种方式：第一种是仅接受允许的数据，这种策略一般是设置防火墙的默认策略为拒绝所有，然后有针对性地放开特定的访问；第二种是只防止不允许的数据，这种策略一般是设置防火墙的默认策略为允许所有，只拒绝已知的非法访问数据。从安全效果而言，前一种防火墙策略表现更为优秀，所以这里使用第一种策略来开发防火墙脚本。

首先在使用 iptables 之前敲入以下两条命令：

```
iptables -F      #清空所有规则
iptables -X      #删除所有自定义的链
```

下面开始建立 iptables 防火墙规则。笔者采取的规则是：默认所有的数据都丢弃，仅接受已知的数据包，所以要有针对地打开需要的端口，下面两条命令定义默认全部丢弃数据包。

```
iptables -P INPUT DROP
iptables -P OUTPUT DROP
#-P参数的意思是policy，即策略。
#第一句的意思是：
#输入(INPUT)的数据包默认的策略(-P)是丢弃(DROP)的
#第二句的意思是：
#输出(OUTPUT)的数据包默认的策略(-P)是丢弃(DROP)的
```

进行到这一步它已经是一个有用的防火墙了，只不过没有什么意义——因为这和拔掉网线的概念没有什么不同，而且比没有防火墙更糟糕的是本地数据包都无法通信了。所以，这种类型的防火墙需要一些基本策略来保证一些基本功能可用，所以下面的一些规则也是

需要的：

```
iptables -A INPUT -p icmp --icmp-type any -j ACCEPT
#允许icmp包进入。如果确认不需要icmp通信，此条可以不写
iptables -A OUTPUT -p icmp --icmp any -j ACCEPT
#允许icmp包出去
iptables -A INPUT -s localhost -d localhost -j ACCEPT
#允许本地数据包出
iptables -A OUTPUT -s localhost -d localhost -j ACCEPT
#允许本地数据包入
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
#允许已经建立和相关的数据包进入
iptables -A OUTPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
#允许已经建立和相关的数据包出去
```

现在需要考虑一些特定策略了，这和服务器的应用类型密切相关。如果是一台 Web 服务器的话，典型的需要是能访问 80 端口，但是就目前的策略而言是无法访问的，所以需要允许 80 端口的访问。命令如下：

```
iptables -A INPUT -p tcp --dport 80 -j ACCEPT
```

如果读者认为这样就大功告成的话那就错了，不信可以尝试访问一下，会发现依然打不开 Web 服务器的主页（假设已设置好 Apache 服务，并应用了以上的防火墙规则）。为什么呢？考虑一下计算机是怎么工作的。假设用户电脑是 A，服务器是 B，从 A 发送了一个目的地址为 B、目的端口是 80 的数据包。服务器 B 收到这个数据包时发现该数据包匹配 INPUT 链规则，所以这个包可以正常的进入服务器 B；然后服务器 B 在给 A 汇包时，回包会进入本地的 OUTPUT 链——但是 OUTPUT 链默认是 DROP 所有包的，而且没有定义相关允许策略，回包无法出去，于是造成了整个访问的过程不完整，所以需要下面的命令：

```
iptables -A OUTPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

现在再试试访问 Web 服务器，一定是成功的。这条命令中使用了状态跟踪模块，意思是对已经建立完整的连接的，以及为了维持该连接需要打开的其他连接所产生的相关连接的所有数据，都可以通过防火墙的 OUTPUT 链。但是如果需要允许该服务器访问其他的 Web 服务器，该怎么办呢？只要打开让数据出去的 80 端口就可以了，这需要两条命令，如下所示：

```
iptables -A OUTPUT -p tcp -m state --state NEW --dport 80 -j ACCEPT
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

如果此时尝试使用服务器访问某个域名，比如 www.baidu.com，会发现其实还是不能访问到页面，难道是上面的规则不对么？考虑一下使用域名访问网站需要经历什么过程。对了，域名解析。因为服务器访问该域名之前需要先解析出它的 IP 地址，所以防火墙必须允许域名解析的数据包出去，使用如下的命令就可以了。


```
iptables -A OUTPUT -p udp --dport 53 -j ACCEPT
```

到了举一反三的时候了，如果要访问 https（默认目标端口为 443）的站点，应该打开什么端口呢？这里请读者自己思考尝试。下面还列举了一些常见的需要打开的端口，读者可以参考设置。

```
#由于管理需要ssh到这台服务器，则需要打开22号端口
iptables -A INPUT -p tcp -dport 22 -j ACCEPT
#如果只允许一个固定的ip能ssh到该服务器的话，上面的语句需要改为
iptables -A INPUT -p tcp --dport 22 -s 192.168.1.10 -j ACCEPT
#可能还需要从该服务器ssh到别的服务器
iptables -A OUTPUT -p tcp --dport 22 -j ACCEPT
```

到这里一个简单的 iptables 防火墙就可以使用了，读者可以领悟一下该脚本开发过程中的各个关键点，最重要的是需要了解服务器可以正常工作时对防火墙策略的需求，以及相应的对端口放开 INPUT 和 OUTPUT 链上的策略。最后将整个过程脚本化，如下所示：

```
#!/bin/bash
#DEFINE VARIABLES
HTTP_PORT=80
SECURE_HTTP_PORT=443
SSH_PORT=22
DNS_PORT=53
ALLOWED_IP=192.168.1.10
IPTABLES=/sbin/iptables
#FLUSH IPTABLES
$IPTABLES -F
$IPTABLES -X
#DEFINE DEFAULT ACTION
$IPTABLES -P INPUT DROP
$IPTABLES -P OUTPUT DROP
#DEFINE INPUT CHAINS
$IPTABLES -A INPUT -p icmp --icmp-type any -j ACCEPT
$IPTABLES -A INPUT -s localhost -d localhost -j ACCEPT
$IPTABLES -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
$IPTABLES -A INPUT -p tcp --dport $SSH_PORT -j ACCEPT
#DEFINE OUTPUT CHAINS
$IPTABLES -A OUTPUT -p icmp --icmp any -j ACCEPT
$IPTABLES -A OUTPUT -s localhost -d localhost -j ACCEPT
$IPTABLES -A OUTPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
$IPTABLES -A OUTPUT -p tcp -m state --state NEW --dport $HTTP_PORT -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport $SECURE_HTTP_PORT -j ACCEPT
$IPTABLES -A OUTPUT -p udp --dport $DNS_PORT -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport $SSH_PORT -j ACCEPT
```

1.6.3 网络连通性诊断

网络是一切系统赖以正常工作的基础设施，所以保证主机的网络连通性是一切工作得以开展的前提。由于网络协议和设备所具有的复杂性，很多故障解决起来是有难度的，不

仅需要有相应的知识结构,有时候还需要有丰富的网络经验。但是从大多数情况看,网络故障主要分为硬件故障、软件故障和网络没有被正确的配置这三种原因。硬件故障又主要分为网卡物理损坏、链路故障等,其中网卡物理损坏主要是指网卡设备由于使用中发生电子元件损坏而造成的网卡设备无法继续使用;链路故障很多时候表现为网线或者水晶头在制作过程中出现线路问题,或是由于线路老化等原因造成物理链路断开,从而致使网络无法物理连通;软件主要表现为网卡驱动故障,也就是操作系统对网卡驱动的不兼容,这个问题往往需要通过安装对应的网卡设备驱动来解决。更多情况下,网络不可达主要不是因为可达性问题,而往往是由于网络未正确配置。

为了诊断网络连通性,我们需要使用一些常见的命令,主要有 ping、host、traceroute 等。

ping 程序的目的在于测试另一台主机是否可达,一般来说,如果 ping 不到某台主机,就说明对方主机已经出现了问题,但是不排除链路中防火墙的因素、ping 包被丢弃等原因而造成 ping 不通。ping 命令最简单的使用方式是接收一个主机名或 IP 作为其单一的参数,在按回车键后,执行 ping 命令的主机会向对端主机发送一个 ICMP 的 echo 请求包,对端主机在接收到这个包后会回应一个 ICMP 的 reply 回应包,在 Linux 下 ping 命令并不会主动停止,需要使用 Ctrl+C 组合键来停止, ping 命令将会对发出的请求包和收到的回应包进行计数,这样就能计算网络丢包率。

host 命令是用来查询 DNS 记录的,如果使用域名作为 host 的参数,命令返回该域名的 IP。命令如下所示:

```
[root@localhost ~]# host www.google.com
www.google.com has address 74.125.128.147
www.google.com has address 74.125.128.103
www.google.com has address 74.125.128.99
www.google.com has address 74.125.128.104
www.google.com has address 74.125.128.105
www.google.com has address 74.125.128.106
www.google.com has IPv6 address 2404:6800:4005:c00::67
```

在 IP 包结构中有一个定义数据包生命周期的 TTL (Time To Live) 字段,该字段用于表明 IP 数据包的生命值,当 IP 数据包在网络上传输时,每经过一个路由器该值就减 1,当该值减为 0 时此包就会被路由器丢弃。这种设计可避免出现一些由于某种原因始终无法到达目的地的包不断地在互联网上传递(可以形象地称之为幽灵包),无谓地耗用网络资源。

不过路由器也不是“无声无息”地将 TTL 值为 0 的 IP 包丢弃的,它会同时给发送该 IP 数据包的主机发送一个 ICMP “超时”消息,主机在接收到这个 ICMP 包后就同时能得到该路由的 IP 地址。

根据上面两个特点,人们写了一个检测数据包是如何经由路由器的程序——traceroute,该程序的工作原理如下:它先构造出一个 TTL 值为 1 的数据包发送给目的主机,这个数据包在经由第一个路由器时,路由器先将 TTL 值减 1 变为 0,然后路由器将该 IP 包丢弃,同时给发送一份 ICMP 消息,这样就得到了经过的第一台路由器的 IP 地址;然

后再构造出一个 TTL 值为 2 的数据包，以此类推，就能得到该 IP 包经历的整条链路的路由器 IP。

这里会有一个问题：tracert 如何确认该 IP 包成功地被目的主机接收了呢？因为目的主机即便收到了 TTL 值为 1 的数据包也不会发送 ICMP 通知给源主机的。这时 tracert 所做的工作就是发送一个 UDP 包给目的主机，同时制定该 UDP 接收的端口为主机不可能存在的端口，主机在接收到这样的包后，由于端口不可达，因此会返回一个“端口不可达”的通知，这样就能确认目的主机是否可以接收到数据包。

基于以上的命令和原理，解决网络在故障时采用的步骤总结如下（其中，不管哪一步出现问题，都需要先解决当前的问题才能进行下一步测试，当所有测试都通过则表示所有问题都已解决了）：

- ❑ 第一步是要确认网卡本身是否能正常工作。利用 ping 工具可以确认这点。输入 ping 127.0.0.1，然后看是否能正常 ping 通。这里的 127.0.0.1 被称为主机的回环接口，是 TCP/IP 协议栈正常工作的前提，如果 ping 不通，一般可以认为本机 TCP/IP 协议栈有问题，但出现这种现象的概率比较低。
- ❑ 第二步是要确认网卡是否出现了物理或驱动故障，使用 ping 本机 IP 地址的方式，如果能 ping 通则说明本地设备和驱动都正常。
- ❑ 第三步要确认是否能 ping 通同网段的其他主机。这一步主要是确认二层网络设备（比如交换机或者 hub）工作是否正常。如果 ping 不通往往说明二层网络上出现了问题，可能涉及交换机的端口工作模式、vlan 划分等因素。
- ❑ 第四步要确认是否能 ping 通网关 IP。如果数据包能正常到达网关，则说明主机和本地网络都工作正常。
- ❑ 第五步确认是否能 ping 通公网上的 IP，如果可以说明本地路由设置正确，否则就要确认路由设备是否做了正确的 nat 或路由设置。
- ❑ 第六步确认是否能 ping 通公网上的某个域名，如果能 ping 通则说明 DNS 部分设置正确。

即便实际工作中可能会受到诸如更复杂的网络环境、安全 ACL、防火墙等众多因素的影响而使网络排查的困难增大，但以上步骤是排除网络故障的主体躯干，在排除不同的网络之间个性化的设置之外，排查的主要步骤都大同小异。

1.7 进程管理

进程是 Linux 系统中一个非常重要的概念，虽然我们无需了解这些进程是如何运行的、内核是如何管理调度的、时间片是如何轮转分配的等，但是需要知道如何控制这些进程，包括查看、启动、关闭、设置优先级等，从而完成好系统工程师的本职工作。

1.7.1 什么是进程

进程表示程序的一次执行过程，它是应用程序的运行实例，是一个动态的过程。或者可以更简单地描述为：进程是操作系统当前运行的程序。当一个进程开始运行时，就是启动了这个过程。进程包括动态执行的程序和数据两部分。在现代操作系统中，支持多进程处理，这些进程可接受操作系统的调度，所以说每一个进程都是操作系统进行资源调度和分配的一个独立单位。

1.7.2 进程的常见状态

所有的进程都可能存在三种状态：运行态、就绪态和阻塞态。运行态表示程序当前实际占用着 CPU 等资源；就绪态是指程序除 CPU 之外的一切运行资源都已经就绪，等待操作系统分配 CPU 资源，只要分配了 CPU 资源，即可立即运行；阻塞态是指程序在运行的过程中由于需要请求外部资源（例如 I/O 资源、打印机等低速或同一时刻只能独享的资源）而当前无法继续执行，从而主动放弃当前 CPU 资源转而等待所请求资源。

进程之间，又存在互斥和同步的关系。互斥即进程间不能同时运行，必须等待一个进程运行完毕，另一个进程才能运行。而进程同步指的是进程间通过某种通信机制实现信息交互。现代计算机使用信号量机制来实现进程间的互斥和同步，它的基本原理是：两个或者多个进程可以通过简单的信号进行合作，一个进程可以被迫在某一位置停止，直到它接收到一个特定的信号。任何复杂的合作需求都可以通过适当的信号结构得到满足。

1.7.3 进程优先级的调整

使用 `top` 命令显示系统运行的程序状态，其中的 `NI` 字段标记了对应进程的优先级，该字段的取值范围是 $-20 \sim 19$ ，数值越低优先级越高，能更多地被操作系统调度运行，如果一个进程在启动时并没有设定 `nice` 优先级，则默认使用 0。普通用户也可以给自己的进程设定 `nice` 优先级，但是范围只限于 $0 \sim 19$ 。`top` 中还有一个 `PR` 字段，它也是进程的“优先级”，这两个概念怎么理解呢？实际上，Linux 使用了“动态优先级”的调度算法来确定每一个进程的优先级。

一个进程的最终优先级 = 优先级 + `nice` 优先级

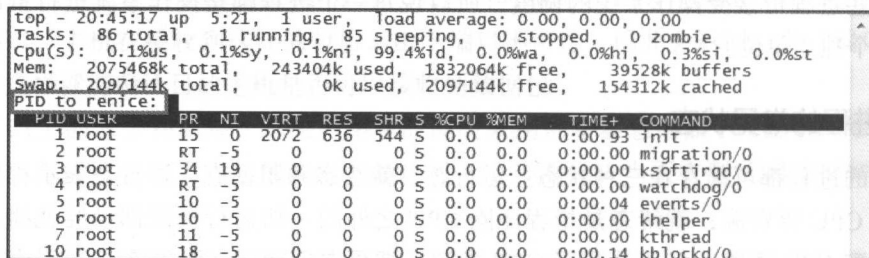
`nice` 命令仅限于在启动一个进程的时候同时赋予其 `nice` 优先级，例如用户写了一个脚本 `job.sh`，想以比较高的优先级来运行它，就可以这么做：

```
[root@localhost ~]# nice -n -10 ./job.sh
```

对于已经启动的进程，可以用 `renice` 命令进行修改，不过，这需要先查询出该进程的 PID（使用 `ps` 命令），假设现在需要将 PID 为 5555 的进程的 `nice` 优先级调整为 -10 ，则可以这么做：

```
[root@localhost ~]# renice -10 -p 5555
```

除了使用 `renice` 外，还可以使用 `top` 提供的功能来修改，前提也是要查到该进程的 PID，然后在 `top` 界面中按 `r` 键，在出现的“PID to renice”后输入 PID（如图 1-55 所示），然后在出现的“renice PID *** to value”后输入修改后的 nice 优先级既可（如图 1-56 所示）。

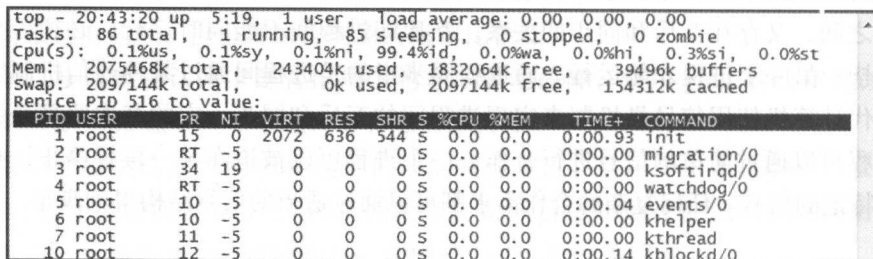


```
top - 20:45:17 up 5:21, 1 user, load average: 0.00, 0.00, 0.00
Tasks: 86 total, 1 running, 85 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.1%us, 0.1%sy, 0.1%ni, 99.4%id, 0.0%wa, 0.0%hi, 0.3%si, 0.0%st
Mem: 2075468k total, 243404k used, 1832064k free, 39528k buffers
Swap: 2097144k total, 0k used, 2097144k free, 154312k cached

PID to renice:
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1	root	15	0	2072	636	544	S	0.0	0.0	0:00.93	init
2	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
4	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
5	root	10	-5	0	0	0	S	0.0	0.0	0:00.04	events/0
6	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	khelper
7	root	11	-5	0	0	0	S	0.0	0.0	0:00.00	kthread
10	root	18	-5	0	0	0	S	0.0	0.0	0:00.14	kblockd/0

图 1-55 调整某个 PID 的优先级



```
top - 20:43:20 up 5:19, 1 user, load average: 0.00, 0.00, 0.00
Tasks: 86 total, 1 running, 85 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.1%us, 0.1%sy, 0.1%ni, 99.4%id, 0.0%wa, 0.0%hi, 0.3%si, 0.0%st
Mem: 2075468k total, 243404k used, 1832064k free, 39496k buffers
Swap: 2097144k total, 0k used, 2097144k free, 154312k cached

Renice PID 516 to value:
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1	root	15	0	2072	636	544	S	0.0	0.0	0:00.93	init
2	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
4	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
5	root	10	-5	0	0	0	S	0.0	0.0	0:00.04	events/0
6	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	khelper
7	root	11	-5	0	0	0	S	0.0	0.0	0:00.00	kthread
10	root	12	-5	0	0	0	S	0.0	0.0	0:00.14	kblockd/0

图 1-56 成功调整了进程的优先级

1.7.4 进程的终止

要终止一个进程，可以通过 `kill`、`pkill`、`killall` 等命令来实现。例如有部分进程由于某种原因已经死掉或是工作异常，抑或是要停止一些非关键或非数据业务的进程，这时就需要使用这些命令来终止进程。这些命令的原理都是向内核发送一个系统操作信号和某个进程的标识号，使得内核对指定标识号的进程进行相应的操作。

一般来说，`kill` 命令需要和 `ps` 命令联合使用。原因是 `kill` 后面跟的应该是需要被终止的进程的 PID，典型用法是使用 `ps` 查出进程的 PID，然后使用 `kill` 将其终止。`kill` 的使用方法是：

```
[root@localhost ~]# kill [信号代码] 进程ID
```

假设系统中的 `dhcpd` 进程由于某种原因需要终止，那么首先要查找到该进程的 PID（从下面的输出中可以看到该 PID 为 2877），然后 `kill` 掉这个 PID，完成这个操作后再看 `dhcpd` 进程，就已经不存在了，示例如下：

```
[root@localhost ~]# ps -ef | grep dhcp
root      2877      1  0 18:59 ?        00:00:00 /usr/sbin/dhcpd
#这里找出dhcpd的PID是2877
#有个更快速的方式来寻找进程的PID; 即使用pidof命令
#[root@localhost ~]# pidof dhcpd
#2877
[root@localhost ~]# kill 2877
```

命令 kill 后面可以跟的信号代码一共有 64 种 (如图 1-57 所示), 常用的一般只有 3 个: HUP (1)、KILL (9)、TERM (15), 分别代表重启、强行杀掉、正常结束。

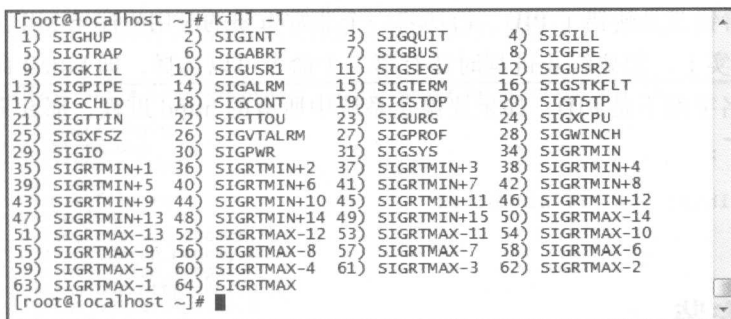


图 1-57 系统中 64 种信号

信号 1 代表重启, 假设需要重启系统中的 httpd 服务, 先查主 httpd 进程的 PID 号, 这里为 2935 (注意, 在图 1-58 中, 第一次查询的时候, 发现有若干个 httpd 进程, 但是主进程只有一个, 即由 root 启动的、PID 为 2935 的第一个进程, 其他的都是该进程的子进程), 使用 kill -1 2935 后, 再查看 httpd 进程的时候, 发现主进程的 PID 没有变化, 而子进程的 PID 都在同一时刻发生了变化, 这说明主进程确实经过了重启。也表明, 使用 kill -1 重启进程的时候实际上是不会改变主进程的 PID 的, 即只是发生了原地重启而已。

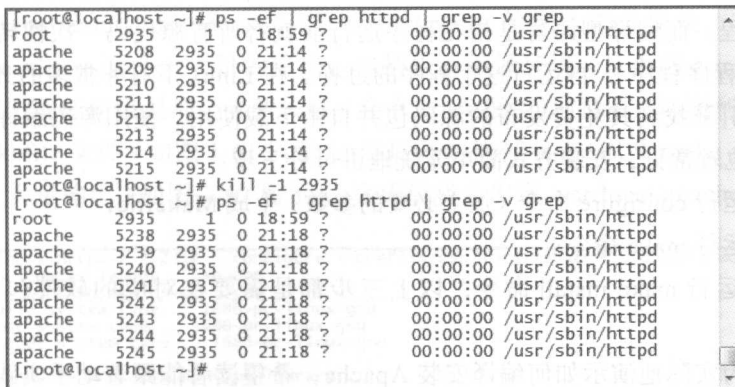


图 1-58 杀掉某个进程

前面成功地使用不带信号代码的 kill 停止了 dhcpd 进程，但实际上有一些进程因为运行中出现问题而无法通过这种方式停止，在这种情况下就需要使用 -9 参数强行停止该进程了，其效果是立即杀死进程，而且该信号无法被阻塞或忽略。但是这个命令也有其天然的危险，即进程直接被系统终止将会导致无法清理之前申请的内存，因此一般情况下不建议使用。而 -15 这个参数就比较温柔了，它会使进程正常退出，它也是 Linux 默认的程序中断信号（也就是在不加参数的情况下默认使用的信号）。

由于使用 kill 命令时要先查询到想要终止的进程的 PID，也就是说操作对象是数字，因此相对来说会比较麻烦，而且在实际的工作中，如果看错了 PID，其后果是无法估计的（想象一下：如果看错或是输错了 PID，恰巧将一个非常重要的应用程序给 kill 了，那就无异于一场灾难）。事实上，想要终止进程时还有第二个命令可以选择，即 killall 命令，它可以直接使用进程的名字而不是 PID，如果要停止系统中所有的 httpd 进程，那么只要按照以下方法操作就可以了：

```
[root@localhost ~]# killall httpd
```

1.8 软件安装

Linux 下安装软件的方式和 Windows 有很大区别，或者说，更为困难。常见的双击进入安装，不断点击“下一步”到底就能完成安装的方式，在 Linux 下很不常见。更多的需要使用命令行的方式进行安装。

1.8.1 源码编译安装

由于计算机不能直接执行用高级语言编写的源程序，因此要想运行代码内容，就要使用一种机制让计算机识别和执行。一般来说，计算机中存在解释型和编译型两种语言。所谓解释型语言，就是计算机逐条取出源码文件的一条指令，将其转化成机器指令，再执行这个指令的过程。而编译型语言是指在程序运行前就将所有源代码一次性转化为机器代码（一般为二进制程序程序），再运行这个过程。在 Linux 下有非常多的开源软件，我们可以通过搜索引擎找到其免费发布的源码包并自由下载使用。使用源码编译安装的方式比较“原始”但也较常见，安装方式简单笼统地讲可分三步：

第一步，运行 configure 命令（加上必要的参数）生成 Makefile；

第二步，运行 make 命令；

第三步，运行 make install 命令，以上三步都是需要在对应的软件包目录根目录中运行。

本节将更为实际地演示如何编译安装 Apache，希望读者能跟着动手实践，以增强对编译安装软件的理解。首先到 Apache 的官方主页 <http://www.apache.org> 下载。这里演示的版

本为 apache-2.2.23, 读者可以根据实际需求下载不同的版本 (如图 1-59 所示)。

```
[root@localhost local]# cd /usr/local/src/
[root@localhost src]# wget http://mirrors.cnnic.cn/apache/httpd/httpd-2.2.23.tar.gz
--2013-02-14 17:07:42-- http://mirrors.cnnic.cn/apache/httpd/httpd-2.2.23.tar.gz
Resolving mirrors.cnnic.cn... 123.125.244.87
Connecting to mirrors.cnnic.cn|123.125.244.87|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7374712 (7.0M) [application/x-gzip]
Saving to: 'httpd-2.2.23.tar.gz'

13% [====>] 986,653 85.7K/s eta 92s
```

图 1-59 下载源码包

下载完成后解压源码包, 并进入该目录 (如图 1-60 所示)。

```
[root@localhost src]# tar zxvf httpd-2.2.23.tar.gz && cd httpd-2.2.23
httpd-2.2.23/
httpd-2.2.23/emacs-style
httpd-2.2.23/httpd.dsp
httpd-2.2.23/libhttpd.dsp
httpd-2.2.23/.deps
httpd-2.2.23/Makefile.in
httpd-2.2.23/include/
httpd-2.2.23/include/scoreboard.h
httpd-2.2.23/include/ap_regkey.h
httpd-2.2.23/include/ap_compat.h
httpd-2.2.23/include/http_config.h
httpd-2.2.23/include/util_time.h
httpd-2.2.23/include/ap_mmn.h
httpd-2.2.23/include/ap_provider.h
```

图 1-60 解压源码包

进入目录后, 需要使用 configure 工具生成 Makefile, 运行 configure 的方式是:

```
[root@localhost httpd-2.2.23]# ./configure --参数1 --参数2...
```

由于配置 Apache 能加的参数非常多, 而且对于新手来说也确实无法分清那么多参数各自的意义 (具体可用参数都可以在 /usr/local/src/httpd-2.2.23/configure 中看到), 这里将介绍用两个比较简单的参数来完成配置的方法。第一个参数是 --prefix=/usr/local/apache/, --prefix, 用于指定安装路径, 一般来说自行编译安装的软件放置的目录建议为 /usr/local/; 第二个参数是 --enable-modules=most, 用于启用 Apache 的绝大部分模块, 非常适合新手使用, 回车后 configure 会产生大量的输出, 包括检查编译环境 (是否有 gcc 工具以及软件依赖关系) 中间出现任何错误都会导致失败 (会有 error 报错), 如果一切顺利, 将会在当前目录下生成 Makefile 文件 (如图 1-61 所示), 然后开始执行 make 以及 make install 命令即可, 此处也会产生大量输出 (如图 1-62 所示), 完成后将会出现 /usr/local/apache 目录。

```
[root@localhost httpd-2.2.23]# ./configure --prefix=/usr/local/apache/ --enable-modules=most
checking for chosen layout... Apache
checking for working mkdir -p... yes
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
Configuring Apache Portable Runtime library ...
```

图 1-61 配置编译参数

```

[root@localhost httpd-2.2.23]# make && make install
Making all in src/lib
make[1]: Entering directory `/usr/local/src/httpd-2.2.23/src/lib'
Making all in apr
make[2]: Entering directory `/usr/local/src/httpd-2.2.23/src/lib/apr'
make[3]: Entering directory `/usr/local/src/httpd-2.2.23/src/lib/apr'
/bin/sh /usr/local/src/httpd-2.2.23/src/lib/apr/libtool --silent --mode=compile gcc -g -O2 -pth
read -DHAVE_CONFIG_H -DLINUX=2 -D_REENTRANT -D_GNU_SOURCE -D_LARGEFILE64_SOURCE -I./includ
e -I/usr/local/src/httpd-2.2.23/src/lib/apr/include/arch/unix -I./include/arch/unix -I/usr/loca
l/src/httpd-2.2.23/src/lib/apr/include/arch/unix -I/usr/local/src/httpd-2.2.23/src/lib/apr/inclu
de -o passwd/apr_getpass.lo -c passwd/apr_getpass.c && touch passwd/apr_getpass.lo
/bin/sh /usr/local/src/httpd-2.2.23/src/lib/apr/libtool --silent --mode=compile gcc -g -O2 -pth
read -DHAVE_CONFIG_H -DLINUX=2 -D_REENTRANT -D_GNU_SOURCE -D_LARGEFILE64_SOURCE -I./includ
e -I/usr/local/src/httpd-2.2.23/src/lib/apr/include/arch/unix -I./include/arch/unix -I/usr/loca
l/src/httpd-2.2.23/src/lib/apr/include/arch/unix -I/usr/local/src/httpd-2.2.23/src/lib/apr/inclu
de -o strings/apr_cpystern.lo -c strings/apr_cpystern.c && touch strings/apr_cpystern.lo
/bin/sh /usr/local/src/httpd-2.2.23/src/lib/apr/libtool --silent --mode=compile gcc -g -O2 -pth
read -DHAVE_CONFIG_H -DLINUX=2 -D_REENTRANT -D_GNU_SOURCE -D_LARGEFILE64_SOURCE -I./includ
e -I/usr/local/src/httpd-2.2.23/src/lib/apr/include/arch/unix -I./include/arch/unix -I/usr/loca

```

图 1-62 编译并安装

安装完成后，使用以下命令启动 Apache 服务，并查看一下 80 端口确实已经被占用。

```
[root@localhost ~]# /usr/local/apache/bin/apachectl start
```

```
[root@localhost ~]# lsof -i:80
```

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE	NODE	NAME
httpd	7149	root	3u	IPv6	59986		TCP	*:http (LISTEN)
httpd	7150	daemon	3u	IPv6	59986		TCP	*:http (LISTEN)
httpd	7151	daemon	3u	IPv6	59986		TCP	*:http (LISTEN)
httpd	7152	daemon	3u	IPv6	59986		TCP	*:http (LISTEN)
httpd	7153	daemon	3u	IPv6	59986		TCP	*:http (LISTEN)
httpd	7154	daemon	3u	IPv6	59986		TCP	*:http (LISTEN)

最后，使用浏览器访问一下服务器的 IP（使用 ifconfig 命令查看服务器 IP），如果看到页面中显示 “It Works” 界面，说明安装成功了。

1.8.2 使用包管理 Yum

Yum（全称为 Yellow dog Updater, Modified）是一个基于 RPM 的 shell 前端包管理器，能够从指定的服务器上（一个或多个）自动下载并安装或更新软件、删除软件，其最大的优点是自动解决依赖关系。

使用 yum 来安装 httpd，只需要使用命令 yum install httpd 即可，在开始的部分打印出的 “Resolving Dependency” 后面就是 yum 首先检查出安装 httpd 时需要安装的依赖包，可以看出这里需要安装 apr 和 apr-util 这两个包，如图 1-63 所示。为什么之前用 RPM 进行安装的时候依赖包比这里多呢？那是因为之前在使用 rpm 包安装 Apache 时已经安装了必要的依赖包，这里使用 yum 进行安装的时候已经满足依赖关系了，所以这里只需要再安装缺失的 apr 包就可以了。

除了安装包之外，yum 也可以删除已经安装的包，如果想删除 httpd 包，只需使用 yum remove httpd 命令即可。

```
[root@localhost ~]# yum install httpd
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
* addons: mirrors.163.com
* base: mirrors.163.com
* extras: mirrors.163.com
* updates: mirrors.163.com
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package httpd.i386 0:2.2.3-76.el5.centos set to be updated
--> Processing Dependency: libapr-1.so.0 for package: httpd
--> Processing Dependency: libaprutil-1.so.0 for package: httpd
--> Running transaction check
--> Package apr.i386 0:1.2.7-11.el5_6.5 set to be updated
--> Package apr-util.i386 0:1.2.7-11.el5_5.2 set to be updated
--> Finished Dependency Resolution

Dependencies Resolved
```

Package	Arch	Version	Repository	Size
Installing:				
httpd	i386	2.2.3-76.el5.centos	updates	1.2 M
Installing for dependencies:				
apr	i386	1.2.7-11.el5_6.5	base	124 k
apr-util	i386	1.2.7-11.el5_5.2	base	80 k

```
Transaction Summary
Install      3 Package(s)
Upgrade     0 Package(s)

Total download size: 1.4 M
Is this ok [y/N]: y
```

图 1-63 使用 yum 安装 httpd

1.8.3 创建自己的 Yum 仓库

创建自己的 Yum 仓库时，可采用如下步骤：

- 1) 安装 Apache 服务（提供 http 协议的共享源）；
- 2) 将安装介质中的内容共享出来；
- 3) 在客户机上配置对应的 repo 文件（repo 文件的内容需要根据源的内容做相应的调整）。

首先演示使用 CentOS 作为源服务器的场景（该服务器的 IP 为 192.168.61.130）。第一步，安装 Apache，该步骤请读者自行完成（使用 RPM 或者 yum 安装，安装完成后启动 httpd 服务）。安装完成后，默认 Apache 的文档目录是 /var/www/html，访问光盘安装介质中的文件，可以用两种方式，一种方式是把 /misc/cd 目录中的所有文件拷贝到 /var/www/html 中，还有更为简单的一种方式：做软链接，示例如下。

```
[root@localhost ~]# cd /var/www/html
[root@localhost html]# ln -s /misc/cd/ .

[root@localhost html]# ls -l #看到软连接已经做好了
total 0
lrwxrwxrwx 1 root root 9 Feb 25 21:06 cd -> /misc/cd/
```

使用浏览器访问该服务器的 `http://IP/cd` 来测试 Apache 是否成功共享安装文件, 如果一切正常, 应该看到如图 1-64 所示的界面。

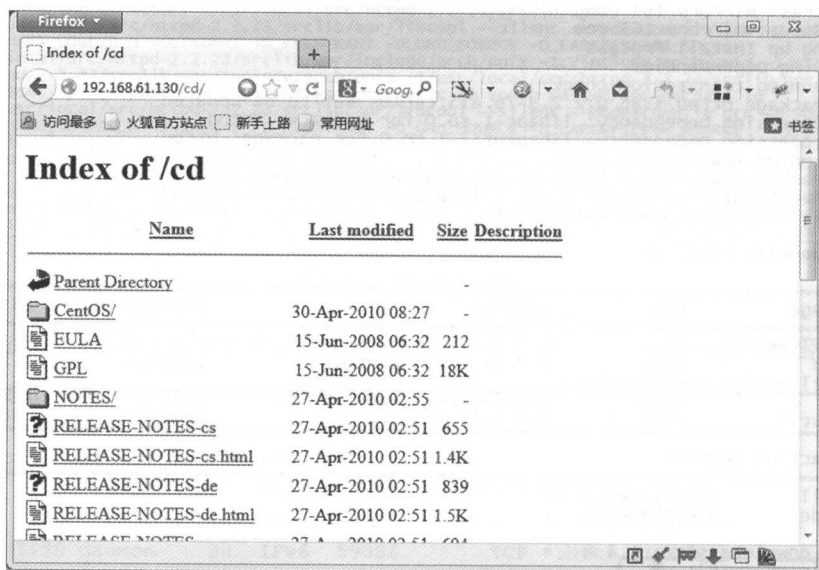


图 1-64 访问 httpd 服务器

至此源服务器就布置好了, 接下来用一台服务器作为客户端测试是否可以使用 (客户端服务器为 RedHat 系统, IP 为 192.168.61.131)。按照如下方式创建 `FirstYum.repo`, 然后更新一下 Yum 缓存, 如果成功的话, 就可以看到如图 1-65 所示的界面, 也能够成功安装软件, 注意到软件来自图 1-66 所示的 `FirstYum`, 这就说明自制的 Yum 仓库成功工作了。

```
[root@localhost yum.repos.d]# cat FirstYum.repo
[FirstYum]
name=FirstYum
baseurl=http://192.168.61.130/cd
gggcheck=1
gggkey=http://192.168.61.130/cd/RPM-GPG-KEY-CentOS-5
```

```
[root@localhost yum.repos.d]# yum clean all && yum makecache
Loaded plugins: rhnplugin, security
Cleaning up Everything
Loaded plugins: rhnplugin, security
This system is not registered with RHN.
RHN support will be disabled.
FirstYum
FirstYum/filelists          1.1 kB    00:00
FirstYum/other              2.9 MB    00:00
FirstYum/group              9.1 MB    00:00
FirstYum/primary            920 kB    00:00
FirstYum                    920 kB    00:00
FirstYum                    2599/2599
FirstYum                    2599/2599
FirstYum                    2599/2599
Metadata Cache Created
[root@localhost yum.repos.d]#
```

图 1-65 从 FirstYum 中更新软件列表

```
[root@localhost ~]# yum install httpd
Loaded plugins: rhnplugin, security
This system is not registered with RHN.
RHN support will be disabled.
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package httpd.i386 0:2.2.3-43.el5.centos set to be updated
--> Processing Dependency: libapr-1.so.0 for package: httpd
--> Processing Dependency: libaprutil-1.so.0 for package: httpd
--> Running transaction check
--> Package apr.i386 0:1.2.7-11.el5_3.1 set to be updated
--> Package apr-util.i386 0:1.2.7-11.el5 set to be updated
--> Processing Dependency: libpq.so.4 for package: apr-util
--> Running transaction check
--> Package postgresql-libs.i386 0:8.1.18-2.el5_4.1 set to be updated
--> Finished Dependency Resolution

Dependencies Resolved
```

Package	Arch	Version	Repository	Size
Installing: httpd	i386	2.2.3-43.el5.centos	FirstYum	1.2 M
Installing for dependencies:				
apr	i386	1.2.7-11.el5_3.1	FirstYum	123 k
apr-util	i386	1.2.7-11.el5	FirstYum	80 k
postgresql-libs	i386	8.1.18-2.el5_4.1	FirstYum	196 k

```
Transaction Summary
Install      4 Package(s)
Upgrade     0 Package(s)

Total download size: 1.6 M
Is this ok [y/N]:
```

图 1-66 成功从 FirstYum 中安装 httpd

1.9 系统安全检测与审计

现在的企业环境都是通过 VPN 和防火墙两道设备将自己包裹在里面，防止外部的入侵攻击，但这只能防止外部的攻击，当出现内部攻击时候，VPN 和防火墙无能为力，所以系统本身带有入侵检测和安全审计两个模块，可从系统本身防范此类问题。

1.9.1 AIDE 系统入侵检测

AIDE (Advanced Intrusion Detection Environment) 是系统自带的一个入侵检测工具，主要目的是检查文件一致性，包括文件是否被更改，文件属性是否变化，文件被修改的时间等。一旦出现 AIDE 监控的文件被篡改的情况，AIDE 会触发告警，通知系统管理员。下面来看如何配置 AIDE。

1. 安装 AIDE 的包

安装命令如下：

```
[root@localhost ~]# yum install aide
```

可以看到配置文件中已经包含了一些默认的规则。如果觉得默认的规则中的监控粒度不够，也可以酌情追加在 NORMALDIR 和 PERMS 中。


```
[root@localhost ~]# cat /etc/aide.conf | grep -A20 'These'

# These are the default rules.
#
#p:      permissions
#i:      inode:
#n:      number of links
#u:      user
#g:      group
#s:      size
#b:      block count
#m:      mtime
#a:      atime
#c:      ctime
#S:      check for growing size
#acl:      Access Control Lists
#selinux  SELinux security context
#xattrs:   Extended file attributes
#md5:      md5 checksum
#sha1:     sha1 checksum
#sha256:   sha256 checksum
#sha512:   sha512 checksum
#rmd160:   rmd160 checksum

# NORMAL = R+rmd160+sha256+whirlpool
NORMAL = R+rmd160+sha256

# For directories, don't bother doing hashes
DIR = p+i+n+u+g+acl+selinux+xattrs

# Access control only
PERMS = p+i+u+g+acl+selinux
```

2. 配置监控规则

这里以 /etc/shadow 为例。将 /etc/aide.conf 配置文件中 88 行向后部分全部注释掉，然后在文件的默认写入如下规则：

```
/etc/shadow    NORMAL
```

3. 测试

AIDE 根据配置文件生成初始化的数据库。然后在系统添加一个 user，此时使用 aide - check 就会看到 AIDE 检测到这个文件变化了，打印出变化的详情。

```
[root@localhost ~]# aide --init

AIDE, version 0.14

### AIDE database at /var/lib/aide/aide.db.new.gz initialized.

[root@localhost ~]# useradd testuser
```

```
[root@localhost ~]# aide --check
File /etc/shadow in databases has different attributes, 340205bbd,240205bbd
AIDE found differences between database and filesystem!!
Start timestamp: 2015-08-03 00:07:44
```

Summary:

```
Total number of files:      4
Added files:                 0
Removed files:               0
Changed files:                2
```

Changed files:

```
changed: /etc/shadow
changed: /etc/shadow-
```

Detailed information about changes:

File: /etc/shadow

```
Size      : 851 , 882
Mtime     : 2015-08-02 17:35:49 , 2015-08-03 00:07:39
Ctime     : 2015-08-02 17:35:49 , 2015-08-03 00:07:39
Inode     : 394023 , 394802
MD5       : 1D5W6SHBeJlJtri5qDCOaA== , /ummfboToD0wfdH/g/+PUg==
RMD160    : yDGi0dfKRWTuhFU+FoBMCU6dlTY= , UNP4d7w+PpJuINW4qGw8dSNAOdE=
SHA256    : R8ur7INKJdJRkYcCdBqz9n0XR885uXwg , 9lJGd00IhWUaq0j8K3Wmp7lXjw+tYOP5
SELinux   : system_u:object_r:shadow_t:s0 , <NULL>
```

File: /etc/shadow-

```
Size      : 725 , 851
Mtime     : 2015-08-02 17:35:45 , 2015-08-02 17:35:49
Ctime     : 2015-08-02 17:35:48 , 2015-08-03 00:07:39
MD5       : 7UIQmn5osGBtiQcPHGz3cQ== , 1D5W6SHBeJlJtri5qDCOaA==
RMD160    : a5Te4JQ3ppdQRh6lTD24gdM+3sM= , yDGi0dfKRWTuhFU+FoBMCU6dlTY=
SHA256    : BVm7gwaNKd4iYtdxQ+0DKnSpQIujcqbZ , R8ur7INKJdJRkYcCdBqz9n0XR885uXwg
```

1.9.2 审计

AIDE 针对的方向是文件完整性，而对于一些系统的操作，可以用系统中自带的 audit 服务来帮助记录以及告警。下面介绍如何配置。

首先，确认 audit 服务是否开启。

```
[root@localhost ~]# /etc/init.d/auditd status
```

然后添加一条规则在 `auditd` 服务的配置文件中，监测 `/mnt` 目录中文件变化的动作，之后重启服务使配置生效。

```
[root@localhost ~]# cat /etc/audit/audit.rules
# This file contains the auditctl rules that are loaded
# whenever the audit daemon is started via the initscripts.
# The rules are simply the parameters that would be passed
# to auditctl.

# First rule - delete all
-D
-w /mnt -p wa -k "config-change"
# Increase the buffers to survive stress events.
# Make this bigger for busy systems
-b 320
```

Feel free to add below this line. See auditctl man page

```
[root@localhost ~]# /etc/init.d/auditd restart
```

```
Stopping auditd:
```

```
[ OK ]
```

```
Starting auditd:
```

```
[ OK ]
```

此时可以看到规则已经生效，在 `/mnt` 目录中创建一个文件，然后更改这个文件的权限，使用 `ausearch` 就可以看到此时审计到的结果。

```
[root@localhost ~]# auditctl -l
-w /mnt/ -p wa -k "config-change"
[root@localhost ~]# touch /mnt/testfile
[root@localhost ~]# chmod 400 /mnt/testfile
[root@localhost ~]# ausearch --start today -k "config-change" -i
----
type=PATH msg=audit(08/03/2015 00:24:06.773:406) : item=1 name=/mnt/testfile inode=
262159 dev=08:02 mode=file,644 ouid=rootogid=root rdev=00:00 nametype=CREATE
type=PATH msg=audit(08/03/2015 00:24:06.773:406) : item=0 name=/mnt/ inode=262147
dev=08:02 mode=dir,755 ouid=root ogid=root rdev=00:00 nametype=PARENT
type=CWD msg=audit(08/03/2015 00:24:06.773:406) : cwd=/root
type=SYSCALL msg=audit(08/03/2015 00:24:06.773:406) : arch=x86_64 syscall=open
success=yes exit=3 a0=0x7ffed732927 a1=O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK
a2=0666 a3=0x3b5dd8f14c items=2 ppid=27507 pid=27715 auid=root uid=root
gid=root euid=rootsuid=root fsuid=root egid=root sgid=root fsgid=root
tty=pts0 ses=47 comm=touch exe=/bin/touch key="config-change"
----
type=PATH msg=audit(08/03/2015 00:24:15.158:407) : item=0 name=/mnt/testfile inode=
262159 dev=08:02 mode=file,644 ouid=rootogid=root rdev=00:00 nametype=NORMAL
type=CWD msg=audit(08/03/2015 00:24:15.158:407) : cwd=/root
type=SYSCALL msg=audit(08/03/2015 00:24:15.158:407) : arch=x86_64 syscall=fchmodat
success=yes exit=0 a0=0xffffffffffff9ca1=0x8150f0 a2=0400 a3=0x0 items=1
ppid=27507 pid=27716 auid=root uid=root gid=root euid=root suid=root fsuid=root
egid=root sgid=root fsgid=root tty=pts0 ses=47 comm=chmod exe=/bin/chmod
key="config-change"
```

或者可以在日志中查找到这个操作的记录:

```
[root@localhost ~]# tail -f /var/log/audit/audit.log
type=PATH msg=audit(1438532646.773:406): item=1 name="/mnt/testfile" inode=262159
type=PATH msg=audit(1438532655.158:407): item=0 name="/mnt/testfile" inode=262159
dev=08:02 mode=0100644 ouid=0 ogid=0 rdev=00:00 nametype=NORMAL
```

系统性能分析

2.1 性能分析简介

很多人喜欢把系统性能分析称为性能优化，这里特意避免使用“优化”一词，是因为优化是一个复杂的、基于业务场景的工作，有时候看似不正常的系统性能现象也可能是正常的表现；有时候看似做了很多针对性的参数调整，但是实际效果可能不如硬件性能提升来得明显。所以本章并不会重点讲解如何优化，而是重点讲解如何分析系统性能。

一般在条件有限的情况下，性能分析主要集中在两个方面：

- ☐ 响应时间
- ☐ 单位时间效率

本章将通过分析系统 CPU、磁盘、内存来讲解寻找系统与应用热点与瓶颈。

2.2 系统分析的基本工具

2.2.1 CPU 性能分析工具

1. mpstat

mpstat 是报告 CPU 状态的工具，用法比较简单，基本用法如下。

(1) 每 1 秒统计一次 CPU 状态，一共统计 3 次

示例代码如下：

```
[root@server ~]# LANG=c  
[root@server ~]# mpstat 1 3
```

```
Linux 3.10.0-123.el7.x86_64 (server.example.com)      05/28/15      _x86_64_
(1 CPU)
```

```
02:00:06      CPU      %usr      %nice      %sys %iowait      %irq      %soft      %steal      %guest
      %gnice      %idle
02:00:07      all      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
02:00:08      all      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
02:00:09      all      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
Average:      all      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
```

在上面的代码中, LANG=c 的目的是将时间从 12 小时制转换成 24 小时制。

(2) 查看多核 CPU 的使用情况

示例代码如下:

```
[root@server ~]# mpstat -P ALL 1 1
```

```
Linux 3.10.0-123.el7.x86_64 (server.example.com)      05/28/2015      _x86_64_
(4 CPU)
```

```
02:07:26 AM      CPU      %usr      %nice      %sys %iowait      %irq      %soft      %steal      %guest
      %gnice      %idle
02:07:27 AM      all      0.25      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 99.75
02:07:27 AM      0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
02:07:27 AM      1      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
02:07:27 AM      2      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
02:07:27 AM      3      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00

Average:      CPU      %usr      %nice      %sys %iowait      %irq      %soft      %steal      %guest
      %gnice      %idle
Average:      all      0.25      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 99.75
Average:      0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
Average:      1      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
Average:      2      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
Average:      3      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
0.00 100.00
```

其中的每一列代表的含义如下:

□ %user: 用户态程序

- ❑ %nice: 优先级调整
- ❑ %sys: 内核态消耗
- ❑ %iowait: 磁盘等待
- ❑ %irp: 硬件中断
- ❑ %soft: 软件中断
- ❑ %steal: 处理 hypervisor 的消耗
- ❑ %guest: 虚拟机消耗掉的 CPU
- ❑ %idle: CPU 空闲

更多的解释请查看 man 手册。

2. 查看 CPU 硬件信息的工具

(1) lscpu

lscpu 这个命令是在 CentOS 6 中引入的, 在 CentOS 5 上没有此工具。lscpu 可以查看 CPU 的型号、一级缓存、二级缓存等信息。示例代码如下:

```
[root@server ~]# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
Stepping:              9
CPU MHz:               3901.000
BogoMIPS:              7802.00
Virtualization:        VT-x
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-3
```

(2) dmidecode

在 CentOS 5 上查看 CPU 硬件信息可以使用 dmidecode 工具, dmidecode 会提供比 lscpu 更为详细的细节信息。示例代码如下:

```
[root@server ~]# dmidecode -t processor | less
# dmidecode 2.12
SMBIOS 2.4 present.
```

Handle 0x0004, DMI type 4, 35 bytes

Processor Information

Socket Designation: CPU socket #0
Type: Central Processor
Family: Unknown
Manufacturer: GenuineIntel
ID: A9 06 03 00 FF FB AB 1F
Version: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
Voltage: 3.3 V
External Clock: Unknown
Max Speed: 30000 MHz
Current Speed: 3900 MHz
Status: Populated, Enabled
Upgrade: ZIF Socket
L1 Cache Handle: 0x0094
L2 Cache Handle: 0x0095
L3 Cache Handle: Not Provided
Serial Number: Not Specified
Asset Tag: Not Specified
Part Number: Not Specified

Handle 0x0005, DMI type 4, 35 bytes

Processor Information

Socket Designation: CPU socket #1
Type: Central Processor
Family: Unknown
Manufacturer: GenuineIntel
ID: A9 06 00 00 FF FB AB 1F
Version: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
Voltage: 3.3 V
External Clock: Unknown
Max Speed: 30000 MHz
Current Speed: 3900 MHz
Status: Populated, Enabled
Upgrade: ZIF Socket
L1 Cache Handle: 0x0096
L2 Cache Handle: 0x0097
L3 Cache Handle: Not Provided
Serial Number: Not Specified
Asset Tag: Not Specified
Part Number: Not Specified

Handle 0x0006, DMI type 4, 35 bytes

.....

2.2.2 内存性能分析工具

1. free

free 是所有系统工程师都会使用的命令，这里要搞清楚 cache、buffer、used 和 total 之间的关系。

首先 $\text{total} = \text{used} + \text{free}$ ，这很容易理解。

□ total: 物理内存的总大小。

□ used: 被使用的内存大小。

□ free: 未被使用的内存大小。

以下是 free 输出的结果：

```
[root@server ~]# free -m
```

	total	used	free	shared	buffers	cached	
Mem:		980	337	643	0	15	91
-/+ buffers/cache:			230	750			
Swap:	1023		0	1023			

在 Mem：这行中的 buffers 和 caches 指的是系统已经分配但是还未被使用的 buffers 和 caches。这里为 $15 + 91 = 106$ ，所以共有 106MB 的 cache/buffer 还未被使用。

-/+ buffers/cache 行中，used 这列代表实际使用的 buffer/cache 总量，即 $337 - 230 = 107$ ，约等于前面的 106。free 这列代表的是系统真正可以使用的内存。

关于 cache 与 buffer 的区别，在后面的章节会讲到。

2. /proc/meminfo

meminfo 里包含了所有的内存相关信息。示例代码如下：

```
[root@server ~]# cat /proc/meminfo
```

MemTotal:	1519556 kB
MemFree:	1132324 kB
MemAvailable:	1192320 kB
Buffers:	1120 kB
Cached:	169944 kB
SwapCached:	0 kB
Active:	124640 kB
Inactive:	139784 kB
Active(anon):	93992 kB
Inactive(anon):	8376 kB
Active(file):	30648 kB
Inactive(file):	131408 kB
Unevictable:	0 kB
Mlocked:	0 kB
SwapTotal:	2113532 kB
SwapFree:	2113532 kB
Dirty:	0 kB
Writeback:	0 kB

```

AnonPages:          93468 kB
Mapped:             30460 kB
Shmem:              9008 kB
Slab:               50548 kB
SReclaimable:       21124 kB
SUnreclaim:         29424 kB
KernelStack:        4920 kB
PageTables:         8332 kB
NFS_Unstable:        0 kB
Bounce:              0 kB
WritebackTmp:        0 kB
CommitLimit:        2873308 kB
Committed_AS:        411276 kB
VmallocTotal:       34359738367 kB
VmallocUsed:         187720 kB
VmallocChunk:       34359533052 kB
HardwareCorrupted:   0 kB
AnonHugePages:       22528 kB
HugePages_Total:     0
HugePages_Free:      0
HugePages_Rsvd:      0
HugePages_Surp:      0
Hugepagesize:        2048 kB
DirectMap4k:         65408 kB
DirectMap2M:        1507328 kB

```

其中一些参数的含义会在后面的章节提到。

3. vmstat

可以说 vmstat 是所有系统管理员必会的命令之一，vmstat 的用法与 mpstat 类似。但是 vmstat 提供了非常丰富的系统信息。因此需要对输出内容有很清楚的了解。下面将讲解几个重点输出，示例如下：

```

[root@server ~]# vmstat -a 1 5
procs -----memory----- --swap-- -----io----- -system-- -----cpu-----
 r  b   swpd   free   inact active    si   so    bi   bo    in   cs us sy id wa st
 1  0     0 1055176 155716 130904     0    0     4    1    9   15  0  0 100  0  0
 0  0     0 1055144 155716 130980     0    0     0    0   44   80  0  0 100  0  0
 0  0     0 1055144 155716 130980     0    0     0    0   30   50  0  0 100  0  0
 0  0     0 1055144 155716 130980     0    0     0    0   34   54  0  0 100  0  0
 0  0     0 1055144 155716 131000     0    0     0    0   24   39  0  0 100  0  0

```

对于其中部分输出项的说明如下。

(1) procs

在 procs 中，b 这列表示的是不可中断睡眠的进程，这个数值往往与磁盘 I/O 有关。

(2) system

system 这列中有两列，分别是 in 和 cs。

in 代表的是每秒钟的中断次数，包括时钟中断。何为时钟中断？时钟中断指的是系统向 CPU 发出信号，请求处理新的时间片。请求的频率叫做时钟频率。这个参数是在内核中配置的。默认配置是每秒钟 1000 次，相当于 1 毫秒一次。这个参数值出现在 `/boot/config-
{kernel-version}` 中，可以使用 `grep` 命令查看到系统当前的数值，示例代码如下：

```
[root@server ~]# grep HZ /boot/config-2.6.32-431.el6.x86_64
CONFIG_NO_HZ=y
# CONFIG_HZ_100 is not set
# CONFIG_HZ_250 is not set
# CONFIG_HZ_300 is not set
CONFIG_HZ_1000=y
CONFIG_HZ=1000
```

cs 代表的是每秒上下文切换数。何为上下文切换？当 CPU 收到时钟请求去处理下一个时间片里的进程时，即处理下一个进程缓存在 CPU 一级缓存的数据，这就是上下文切换。

in 与 cs 数值偏高说明系统非常繁忙。

(3) CPU

CPU 这部分中，st 这列往往会被很多人忽视，其实这列在虚拟化的环境中是比较重要的。st 全称是 steal time，指的是强制等待虚拟 CPU 的时间，如果这个数值过高，说明 hypervisor 进程正在为别的虚拟机服务，此时需要等待 hypervisor。在生产环境中 st 持续偏高，说明物理主机上运行了太多的虚拟机，已经超出了物理机器的资源。

2.2.3 磁盘性能分析工具

1. iostat

iostat 也是所有系统管理员必会的命令之一，具体使用不多细说，但可能很多人并不清楚 iostat 输出值的单位是什么含义，而这恰恰是非常重要的。

默认情况下 iostat 输出是以 block 为单位的。以 Blk 开头的值都是以 block 为单位的，在 iostat 中，一个 block 是 512 个字节。示例代码如下：

```
[root@server ~]# iostat 1 5 /dev/sda
Linux 2.6.32-431.11.2.el6.x86_64 (server.example.com) 05/28/2015 _x86_64_ (16 CPU)

avg-cpu:  %user   %nice   %system   %iowait   %steal   %idle
           5.62    0.00    4.84      0.80     0.00   88.74

Device:            tps    Blk_read/s    Blk_wrtn/s        Blk_read    Blk_wrtn
sda                24.02        442.14         683.54    14196546071    21947219860

avg-cpu:  %user   %nice   %system   %iowait   %steal   %idle
           6.92    0.00    4.85      0.06     0.00   88.17

Device:            tps    Blk_read/s    Blk_wrtn/s        Blk_read    Blk_wrtn
sda                2.00         0.00        24.00         0         24
```

```
avg-cpu:  %user   %nice   %system   %iowait   %steal   %idle
           6.75    0.00    4.73     0.00    0.00   88.52
```

```
Device:  tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda      0.00         0.00         0.00         0          0
```

```
avg-cpu:  %user   %nice   %system   %iowait   %steal   %idle
           8.12    0.00    4.91     0.38    0.00   86.60
```

```
Device:  tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda      5.00         0.00       112.00         0        112
```

```
avg-cpu:  %user   %nice   %system   %iowait   %steal   %idle
           2.46    0.00    4.93     1.77    0.00   90.84
```

```
Device:  tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda     42.00         0.00       528.00         0        528
```

所以如果希望以 KB 形式显示, 需要加上 -k 参数将其转换成字节, 如下:

```
[root@server ~]# iostat 1 5 -k /dev/sda
```

```
Linux 2.6.32-431.11.2.el6.x86_64 (server.example.com) 05/28/2015 _x86_64_ (16 CPU)
```

```
avg-cpu:  %user   %nice   %system   %iowait   %steal   %idle
           5.62    0.00    4.84     0.80    0.00   88.74
```

```
Device:  tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda     24.02       221.07       341.77   7098281615 10973621830
```

```
avg-cpu:  %user   %nice   %system   %iowait   %steal   %idle
           0.75    0.00    5.75     0.00    0.00   93.50
```

```
Device:  tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda      0.00         0.00         0.00         0          0
```

```
avg-cpu:  %user   %nice   %system   %iowait   %steal   %idle
           0.57    0.00    5.28     0.00    0.00   94.15
```

```
Device:  tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda      4.00         0.00       16.00         0          16
```

```
avg-cpu:  %user   %nice   %system   %iowait   %steal   %idle
           0.76    0.00    4.79     0.50    0.00   93.95
```

```
Device:  tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda      9.00         0.00      100.00         0        100
```

```
avg-cpu:  %user   %nice   %system   %iowait   %steal   %idle
           1.07    0.00    4.91     1.20    0.00   92.82
```



```
Device:  tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda      34.00         0.00        456.00         0        456
```

2. iotop

iotop 是用 Python 写的一个类似于 top 命令的软件，用来监控磁盘 I/O 的情况。iotop 可以实时监控到每个进程及线程的磁盘读写和 I/O 请求。

其中，需要注意以下几个参数：

- ❑ -o：只显示有 I/O 操作的进程和线程。
- ❑ -P：只显示进程数。默认是显示进程和线程。
- ❑ -k：以千字节显示，更为友好的输出。

示例代码如下：

```
[root@server ~]# iotop -h
Usage: /usr/sbin/iotop [OPTIONS]
```

DISK READ and DISK WRITE are the block I/O bandwidth used during the sampling period. SWAPIN and IO are the percentages of time the thread spent respectively while swapping in and waiting on I/O more generally. PRIO is the I/O priority at which the thread is running (set using the ionice command).

Controls: left and right arrows to change the sorting column, r to invert the sorting order, o to toggle the --only option, p to toggle the --processes option, a to toggle the --accumulated option, i to change I/O priority, q to quit, any other key to force a refresh.

Options:

```
--version          show program's version number and exit
-h, --help         show this help message and exit
-o, --only         only show processes or threads actually doing I/O
-b, --batch        non-interactive mode
-n NUM, --iter=NUM number of iterations before ending [infinite]
-d SEC, --delay=SEC delay between iterations [1 second]
-p PID, --pid=PID  processes/threads to monitor [all]
-u USER, --user=USER users to monitor [all]
-P, --processes    only show processes, not all threads
-a, --accumulated  show accumulated I/O instead of bandwidth
-k, --kilobytes    use kilobytes instead of a human friendly unit
-t, --time         add a timestamp on each line (implies --batch)
-q, --quiet        suppress some lines of header (implies --batch)
```

2.2.4 sar

sar 可以说是系统性能诊断的瑞士军刀了，它可以提供几乎所有的系统信息。同时也可通过结合 cronjob、sar 记录下系统的实时状态，以便系统管理员能够对过去的性能状态进行分析排错。

1. 自动收集系统活动信息

在 `/etc/cron.d/sysstat` 里有两个计划任务，`sa1` 收集当前系统的信息，`sa2` 汇集当天的系统信息。在 `cronjob` 里设定了 `sa1` 每 10 分钟运行一次，`sa2` 则在每天的 23 点 59 分运行一次。这里建议采用默认值。示例如下：

```
[root@server ~]# cat /etc/cron.d/sysstat
# Run system activity accounting tool every 10 minutes
*/10 * * * * root /usr/lib64/sa/sa1 1 1
# 0 * * * * root /usr/lib64/sa/sa1 600 6 &
# Generate a daily summary of process accounting at 23:53
53 23 * * * root /usr/lib64/sa/sa2 -A
```

这里，`sa1` 调用了 `sadc` 来收集当前系统的活动信息，以 2 进制形式保存数据。`sadc` 叫做数据收集实用程序，它是 `sar` 的后端程序。

`sa2` 调用了 `sar` 来生成当天的系统信息报告，以文本形式保存。

`sa1` 和 `sa2` 都是 shell 脚本，也是系统管理员学习 `bash` 编程的经典教例。

2. 查看过去的系统活动信息

`sar` 的基本使用方法大家早已熟练，比如使用 `-d` 参数查看磁盘 I/O，`-r` 查看系统内存状态等，本节不再重复，具体参数可以详细阅读 `man` 手册。本节重点讲解查找过去某一时段内系统状态的方法，比如需要查找过去某个时刻系统进程数最高的那个时间点与进程数时，可以读取位于 `/var/log/sa` 中历史的 `sar` 数据，然后利用 `sort` 命令对指定列进行排序，示例代码如下：

```
[root@server ~]# sar -f /var/log/sa/sa22 -q | sort -nr -k 3 | more
Average:      17      878      3.15      4.56      4.93
02:30:01 PM    56    1044      4.20      5.90      6.04
01:20:01 AM    51     908      4.29      5.26      5.67
05:30:01 PM    50     911      2.53      3.03      3.41
08:30:01 AM    46     931      3.48      5.59      8.40
06:40:01 AM    46     913      4.24      5.40      5.72
08:20:01 AM    44     920      4.20      7.34     10.78
08:10:01 AM    44     999      5.50     19.59     15.37
04:30:01 AM    44     919      1.89      2.97      3.24
.....

Linux 2.6.32-279.5.2.el6.x86_64 (server) 05/22/2015 _x86_64_ (24 CPU)
12:00:01 AM runq-sz plist-sz ldavg-1 ldavg-5 ldavg-15
```

`sar -q` 用于显示队列与进程数，从文件中读取的时候，只需要使用 `-f` 参数指定对应的 `sar` 信息文件即可。

第三列 `plist-sz` 是系统的进程数，使用 `sort` 对第三列进行排列就可以找出最大进程数与最大进程数的时间。

2.3 软件分析的基本工具

2.3.1 ldd

ldd 是一个用来查看程序运行所需共享库的工具。它会告诉用户这个程序依赖了哪些库文件、库文件的位置，以及是否缺少库文件等。

ldd 其实只是一个 shell 的脚本，其原理是调用 ld-linux.so 模块来查看程序依赖的共享库。示例代码如下：

```
[root@el6-build ~]# ldd /usr/bin/mysql
linux-vdso.so.1 => (0x00007fff4d7df000)
libncursesw.so.5 => /lib64/libncursesw.so.5 (0x00007f21347eb000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f21345cd000)
libmysqlclient.so.16 => /usr/lib64/mysql/libmysqlclient.so.16 (0x00007f2134249000)
libcrypt.so.1 => /lib64/libcrypt.so.1 (0x00007f2134012000)
libnsl.so.1 => /lib64/libnsl.so.1 (0x00007f2133df8000)
libssl.so.10 => /usr/lib64/libssl.so.10 (0x00007f2133b8d000)
libcrypto.so.10 => /usr/lib64/libcrypto.so.10 (0x00007f21337ae000)
libz.so.1 => /lib64/libz.so.1 (0x00007f2133597000)
libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00007f2133291000)
libm.so.6 => /lib64/libm.so.6 (0x00007f213300d000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f2132df6000)
libc.so.6 => /lib64/libc.so.6 (0x00007f2132a62000)
libtinfo.so.5 => /lib64/libtinfo.so.5 (0x00007f2132841000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f213263c000)
/lib64/ld-linux-x86-64.so.2 (0x00007f2134a24000)
libfreebl3.so => /lib64/libfreebl3.so (0x00007f21323c5000)
libgssapi_krb5.so.2 => /lib64/libgssapi_krb5.so.2 (0x00007f2132181000)
libkrb5.so.3 => /lib64/libkrb5.so.3 (0x00007f2131e9a000)
libcom_err.so.2 => /lib64/libcom_err.so.2 (0x00007f2131c96000)
libk5crypto.so.3 => /lib64/libk5crypto.so.3 (0x00007f2131a6a000)
libkrb5support.so.0 => /lib64/libkrb5support.so.0 (0x00007f213185e000)
libkeyutils.so.1 => /lib64/libkeyutils.so.1 (0x00007f213165b000)
libresolv.so.2 => /lib64/libresolv.so.2 (0x00007f2131441000)
libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f2131221000)
```

2.3.2 strace 与 ltrace

在 Linux 系统中，系统调用（system call）就是内核态给用户态提供的一个系统接口，通过这个接口，可以非常容易地从用户态切换到内核态工作，strace 和 ltrace 就是用于追踪这种系统调用的，strace 与 ltrace 分别用来跟踪进程的系统调用和库函数调用。

下面用一个非常简单的 python 脚本来演示下如何使用 strace 与 ltrace。

首先创建一个 python 脚本，只需要打印 hello world 即可。然后使用这个脚本作为 strace 和 ltrace 的示例。

```
[root@server ~]# cat hello.py
#!/usr/bin/python
print 'hello world!'
```

1. strace

(1) 查看 hello.py 脚本运行过程中系统调用的全过程

查看脚本运行时系统调用的命令非常简单，示例代码如下，从输出中可以看到，在执行这个 python 脚本的过程中，系统在背后做了很多的事情，因为输出太长，这里只选取开头部分。

```
[root@server ~]# strace ./hello.py
execve("./hello.py", ["/usr/bin/python", [/* 22 vars */]) = 0
brk(0) = 0x1a46000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2ef2379000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=69138, ...}) = 0
mmap(NULL, 69138, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2ef2368000
close(3) = 0
open("/lib64/libpython2.7.so.1.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p\363\3\0\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1822488, ...}) = 0
mmap(NULL, 3954184, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f2ef1d94000
mprotect(0x7f2ef1f0c000, 2097152, PROT_NONE) = 0
mmap(0x7f2ef210c000, 258048, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x178000) = 0x7f2ef210c000
mmap(0x7f2ef214b000, 58888, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f2ef214b000
close(3) = 0
open("/lib64/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\240\1\0\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=141616, ...}) = 0
mmap(NULL, 2208864, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f2ef1b78000
mprotect(0x7f2ef1b8e000, 2097152, PROT_NONE) = 0
mmap(0x7f2ef1d8e000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16000) = 0x7f2ef1d8e000
mmap(0x7f2ef1d90000, 13408, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f2ef1d90000
close(3) = 0
open("/lib64/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\320\16\0\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=19512, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
```

```

0x7f2ef2367000
mmap(NULL, 2109744, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f2ef1974000
mprotect(0x7f2ef1977000, 2093056, PROT_NONE) = 0

```

比如 `open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3` 这行, `open` 的这种系统调用可以通过 `man` 手册来查阅, 具体可以查阅 `man syscalls`。

(2) 统计有多少个系统调用

使用 `-c` 参数可以统计出所有的系统调用与调用次数。示例代码如下:

```

[root@server ~]# strace -c ./hello.py
hello world!

```

% time	seconds	usecs/call	calls	errors	syscall
16.31	0.001752	9	186	122	open
15.36	0.001650	28	59		mmap
13.75	0.001477	22	68		rt_sigaction
10.58	0.001136	81	14		mprotect
9.51	0.001021	10	106		read
7.76	0.000833	13	66		close
7.32	0.000786	8	99		fstat
6.01	0.000645	7	89	61	stat
3.67	0.000394	12	32		munmap
3.00	0.000322	11	30		brk
1.31	0.000141	141	1		execve
0.84	0.000090	18	5	1	ioctl
0.74	0.000080	80	1	1	access
0.74	0.000080	80	1		set_tid_address
0.72	0.000077	77	1		set_robust_list
0.66	0.000071	71	1		arch_prctl
0.39	0.000042	21	2		openat
0.34	0.000037	37	1		rt_sigprocmask
0.34	0.000037	37	1		getrlimit
0.34	0.000036	6	6		lstat
0.32	0.000034	11	3		lseek
0.00	0.000000	0	1		write
0.00	0.000000	0	4		getdents
0.00	0.000000	0	1		getcwd
0.00	0.000000	0	6	2	readlink
0.00	0.000000	0	1		getuid
0.00	0.000000	0	1		getgid
0.00	0.000000	0	1		geteuid
0.00	0.000000	0	1		getegid
100.00	0.010741		788	187	total

(3) 按照 calls 的次数排序

如果希望知道 `syscall` 中哪几种 `call` 最多, 可以使用如下代码:

```

[root@server ~]# strace -c -S calls ./hello.py

```

```
hello world!
```

% time	seconds	usecs/call	calls	errors	syscall
10.62	0.000684	4	186	122	open
6.97	0.000449	4	106		read
8.18	0.000527	5	99		fstat
0.57	0.000037	0	89	61	stat
3.74	0.000241	4	68		rt_sigaction
9.56	0.000616	9	66		close
28.12	0.001811	31	59		mmap
3.03	0.000195	6	32		munmap
1.29	0.000083	3	30		brk
18.12	0.001167	83	14		mprotect
0.00	0.000000	0	6		lstat
0.00	0.000000	0	6	2	readlink
0.00	0.000000	0	5	1	ioctl
0.00	0.000000	0	4		getdents
0.00	0.000000	0	3		lseek
0.00	0.000000	0	2		openat
0.00	0.000000	0	1		write
1.34	0.000086	86	1		rt_sigprocmask
1.47	0.000095	95	1	1	access
1.49	0.000096	96	1		execve
0.00	0.000000	0	1		getcwd
1.34	0.000086	86	1		getrlimit
0.00	0.000000	0	1		getuid
0.00	0.000000	0	1		getgid
0.00	0.000000	0	1		geteuid
0.00	0.000000	0	1		getegid
1.57	0.000101	101	1		arch_prctl
1.32	0.000085	85	1		set_tid_address
1.27	0.000082	82	1		set_robust_list
100.00	0.006441		788	187	total

(4) 只看某一种 syscall 的调用情况

下面的代码会使用 `-e` 参数指定系统调用的类型。

```
[root@server ~]# strace -c -e open ./hello.py
hello world!
```

% time	seconds	usecs/call	calls	errors	syscall
100.00	0.002185	12	186	122	open
100.00	0.002185		186	122	total

2. ltrace

`ltrace` 的用法与 `strace` 类似，重点在函数调用方面。

(1) 跟踪库函数的调用

在 `ltrace` 里跟踪库函数的调用可使用 `-cf` 参数，示例代码如下：


```
[root@server ~]# ltrace -cf grep root /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
```

```
operator:x:11:0:operator:/root:/sbin/nologin
```

% time	seconds	usecs/call	calls	function

19.50	0.006624	86	77	malloc
18.01	0.006116	98	62	strlen
12.34	0.004190	59	71	free
8.40	0.002853	95	30	realloc
7.83	0.002660	80	33	__ctype_get_mb_cur_max
7.74	0.002629	97	27	strcpy
7.05	0.002395	95	25	strncmp
6.54	0.002220	69	32	mbrtowc
1.65	0.000562	70	8	calloc
1.14	0.000386	193	2	setlocale
1.12	0.000380	95	4	wcrtomb
0.88	0.000298	74	4	wctob
0.82	0.000278	139	2	read
0.72	0.000246	246	1	re_compile_pattern
0.65	0.000221	55	4	memchr
0.46	0.000157	157	1	__cxa_atexit
0.44	0.000148	74	2	__fpending
0.43	0.000145	72	2	fwrite_unlocked
0.37	0.000124	124	1	getpagesize
0.36	0.000123	123	1	
0.36	0.000122	61	2	fclose
0.34	0.000117	58	2	isatty
0.33	0.000111	111	1	strrchr
0.32	0.000109	109	1	_obstack_begin
0.32	0.000107	107	1	close
0.25	0.000084	84	1	bindtextdomain
0.24	0.000080	80	1	textdomain
0.24	0.000080	80	1	getopt_long
0.23	0.000079	79	1	re_set_syntax
0.23	0.000078	78	1	strcmp
0.22	0.000076	76	1	getenv
0.17	0.000059	59	1	__xstat
0.17	0.000058	58	1	open
0.15	0.000050	50	1	nl_langinfo

100.00	0.033965		405	total

(2) 追踪一个进程的库函数调用

这里以服务器上的一个 mysql 进程为例，首先获取到 mysql 进程的 pid，然后在 ltrace 中使用 -p 参数加上 mysql 的 pid 即可追踪 mysql 这个进程的库函数调用。

```
[root@server ~]# ps -aux | grep mysql
```

```
root      1704  0.0  0.0 106064 1488 pts/0    S      02:04   0:00 /bin/sh /usr/
bin/mysqld_safe --datadir=/var/lib/mysql --socket=/var/lib/mysql/mysql.sock
--pid-file=/var/run/mysqld/mysqld.pid --basedir=/usr --user=mysql
```

```
mysql      1806  0.0  0.6 367948 27288 pts/0    Sl   02:04   0:05 /usr/libexec/
mysql --basedir=/usr --datadir=/var/lib/mysql --user=mysql --log-error=/
var/log/mysql.log --pid-file=/var/run/mysql/mysql.pid --socket=/var/lib/
mysql/mysql.sock
root      29285  0.0  0.0 103312   824 pts/0    S+   11:53   0:00 grep mysql
```

```
[root@server ~]# ltrace -p 1806
```

```
[pid 1813] pthread_mutex_trylock(0x7fe5bfff02920, 0, 0, -1, 0x7fe5bcc3dd30) = 0
[pid 1813] pthread_mutex_unlock(0x7fe5bfff02920, 0, 0, 0x7fe5c81af628,
0x7fe5bcc3dd30) = 0
[pid 1813] time(NULL) = 1432871609
[pid 1813] difftime(0x5567e2b9, 0x5567e28b, 859093, 0x20c49ba5e353f7cf,
0x963e07f8e9ca) = 0x5567e2b9
[pid 1813] pthread_mutex_lock(0x1669070, 0x5567e28b, 859093, 0x20c49ba5e353f7cf,
0x963e07f8e9ca) = 0
[pid 1813] pthread_mutex_unlock(0x1669070, 3, 1, 0x20c49ba5e353f7cf, 0x1669070) = 0
[pid 1813] pthread_mutex_lock(0x1669070, 0x7fe5bcc3dd90, 0x1669070,
0x20c49ba5e353f7cf, 0x1669070) = 0
[pid 1813] pthread_mutex_unlock(0x1669070, 9999, 1, 0x20c49ba5e353f7cf,
0x1669070) = 0
[pid 1813] fflush(0x7fe5c6b4c860) = 0
[pid 1813] select(0, 0, 0, 0, 0x7fe5bcc3dd30 <unfinished ...>
[pid 1812] pthread_mutex_trylock(0x7fe5bfeff2c8, 0, 0, -1, 0x7fe5bd63ed70) = 0
[pid 1812] pthread_mutex_lock(0x19864b0, 0, 0, 0x7fe5c81af628, 0x7fe5bd63ed70) = 0
[pid 1812] pthread_mutex_unlock(0x19864b0, 3, 1, 0x7fe5c81af628, 0x19864b0) = 0
[pid 1812] pthread_mutex_unlock(0x7fe5bfeff2c8, 0, 0x19864b0, 0x7fe5c81af628,
0x19864b0) = 0
[pid 1812] select(0, 0, 0, 0, 0x7fe5bd63ed70 <unfinished ...>
[pid 1813] <... select resumed> ) = 0
[pid 1813] pthread_mutex_trylock(0x7fe5bfff02920, 0, 0, -1, 0x7fe5bcc3dd30) = 0
[pid 1813] pthread_mutex_unlock(0x7fe5bfff02920, 0, 0, 0x7fe5c81af628,
0x7fe5bcc3dd30)
.....
```

2.3.3 ipcs

进程间通信是系统中常见的场景，多个进程可能会需要调用同一个内存内容，比如管道，前一个进程的输出放入内存，后一个命令去读取这段内存。

一共有三种进程间通信方法：

- ❑ semaphores: 表示信号量。
- ❑ message queues: 表示消息队列。
- ❑ share memory regions: 表示共享内存段。

用户可以使用 ipcs 这个命令来查看以上三种进程间通信的具体情况，示例如下：

```
[root@server ~]# ipcs
```

```
----- Message Queues -----
```

```

key          msqid      owner      perms      used-bytes   messages

----- Shared Memory Segments -----
key          shmids      owner      perms      bytes         nattch     status
0x01125aae 0              root       600         1000          9

----- Semaphore Arrays -----
key          semid      owner      perms      nsems
0x00000000 131072     apache     600         1
0x00000000 163841     apache     600         1
0x00000000 196610     apache     600         1
0x00000000 229379     apache     600         1
0x00000000 262148     apache     600         1

```

1. 配置共享内存

一般情况下，系统管理员很少遇到处理共享内存的情况，系统默认的配置已经足够使用，所以这里只做简单的讲解。

假设现在因为一些需求，要限制进程申请的共享内存空间最大 1024MB。

首先，使用 `ipcs -l -m` 查看到现在系统的最大共享内存空间，这里为 1073741824，在 kernel 中这个参数是由 `kernel.shmall` 控制的，`kernel.shmall` 的单位是 page，所以要将 1024MB 转换为 page 数目。使用 `sysctl -w` 可让修改即时生效。示例如下：

```

[root@server ~]# ipcs -l -m
----- Shared Memory Limits -----
max number of segments = 4096
max seg size (kbytes) = 4194303
max total shared memory (kbytes) = 1073741824
min seg size (bytes) = 1

[root@server ~]# echo $[1024*1024/4]
262144
[root@server ~]# sysctl -w kernel.shmall=262144
kernel.shmall = 262144
[root@server ~]# ipcs -l -m

----- Shared Memory Limits -----
max number of segments = 4096
max seg size (kbytes) = 4194303
max total shared memory (kbytes) = 1048576
min seg size (bytes) = 1

```

2. 清除共享内存

清除共享内存也是一个很少会触发的动作，但还是要知道如何使用 `ipcs` 命令查看现在的共享内存段，可使用 `ipcrm` 清除共享内存。

示例如下：

```

[root@server ~]# ipcs -m

```

```
----- Shared Memory Segments -----
```

key	shmid	owner	perms	bytes	nattch	status
0x0112536f	0	root	600	1000	6	

```
[root@server ~]# ipcrm -M 0x0112536f
```

```
[root@server ~]# ipcs -m
```

```
----- Shared Memory Segments -----
```

key	shmid	owner	perms	bytes	nattch	status
0x00000000	0	root	600	1000	6	dest

2.3.4 systemtap

systemtap 是一个非常著名的内核态进程跟踪程序，它的作用非常广泛，最主要的作用是寻找程序的性能瓶颈。

Linux 内核里有一个 kprobe 机制，这是一个动态的收集 debug 信息的工具，systemtap 就是基于 kprobe 机制的一个调试工具。systemtap 的使用简单，而且可以自己定义脚本，对开发人员、系统管理员来说是一个深入分析性能的利器。本节将讲述如何安装配置 systemtap，以及如何使用已有的 systemtap 脚本。

1. 安装准备

systemtap 的安装需要准备一台测试机器，先在测试机器上安装 kernel-debuginfo、kernel-debuginfo-common、kernel-level、systemtap-runtime、gcc 等相关包，然后在测试机器上编译测试脚本，编译测试完成之后将编译好的模块放在生产机器上，生产机器只需要安装 systemtap-runtime 包即可。

CentOS 的 debuginfo 相关安装包可以在 <http://debuginfo.centos.org/> 里找到。

注意，安装 kernel-debuginfo 时，相应的内核版本一定要一致！

2. 安装 kernel 相关包

首先要确定系统内核版本，然后下载相对应的 kernel-debuginfo 包。示例如下：

```
[root@systemtap ~]# uname -a
```

```
Linux systemtap.example.com 2.6.32-504.el6.x86_64 #1 SMP Wed Oct 15 04:27:16 UTC
2014 x86_64 x86_64 x86_64 GNU/Linux
```

```
[root@systemtap ~]# wget http://debuginfo.centos.org/6/x86_64/kernel-debug-
debuginfo-2.6.32-504.el6.x86_64.rpm
```

```
[root@systemtap ~]# wget http://debuginfo.centos.org/6/x86_64/kernel-debuginfo-
2.6.32-504.el6.x86_64.rpm
```

```
[root@systemtap ~]# wget http://debuginfo.centos.org/6/x86_64/kernel-debuginfo-
common-x86_64-2.6.32-504.el6.x86_64.rpm
```

```
[root@systemtap ~]# rpm -ivh kernel-debuginfo-common-x86_64-2.6.32-504.el6.x86_64.rpm
Preparing... ##### [100%]
1:kernel-debuginfo-common##### [100%]
[root@systemtap ~]# rpm -ivh kernel-debuginfo-2.6.32-504.el6.x86_64.rpm
Preparing... ##### [100%]
1:kernel-debuginfo ##### [100%]
[root@systemtap ~]# rpm -ivh kernel-debug-debuginfo-2.6.32-504.el6.x86_64.rpm
Preparing... ##### [100%]
1:kernel-debug-debuginfo ##### [100%]
```

3. 安装 Systemtap 软件包

安装命令如下：

```
[root@systemtap ~]# yum -y install install gcc systemtap systemtap-runtime
kernel-debug kernel-devel kernel-debug-devel kernel-firmware
```

4. 编译 Systemtap 脚本

Systemtap 软件包本身已经提供了一些脚本，这些脚本涉及 I/O、内存、网络等，并且几乎可以直接使用。下面以 profiling 中的 topsys.stp 为例，讲解如何编译。

```
[root@systemtap ~]# cd /usr/share/doc/systemtap-client-2.5/example
[root@systemtap examples]# ls -lrt
total 412
-rw-r--r-- 1 root root 5886 Oct 15 2014 README
-rw-r--r-- 1 root root 91092 Oct 15 2014 keyword-index.txt
-rw-r--r-- 1 root root 140946 Oct 15 2014 keyword-index.html
-rw-r--r-- 1 root root 47261 Oct 15 2014 index.txt
-rw-r--r-- 1 root root 79298 Oct 15 2014 index.html
drwxr-xr-x 3 root root 4096 Jun 1 00:29 general
drwxr-xr-x 2 root root 4096 Jun 1 00:29 html
drwxr-xr-x 2 root root 4096 Jun 1 00:29 interrupt
drwxr-xr-x 2 root root 4096 Jun 1 00:29 io
drwxr-xr-x 2 root root 4096 Jun 1 00:29 locks
drwxr-xr-x 2 root root 4096 Jun 1 00:29 memory
drwxr-xr-x 2 root root 4096 Jun 1 00:29 network
drwxr-xr-x 2 root root 4096 Jun 1 00:29 process
drwxr-xr-x 2 root root 4096 Jun 1 00:29 profiling
drwxr-xr-x 3 root root 4096 Jun 1 00:29 stappgames
drwxr-xr-x 2 root root 4096 Jun 1 00:29 virtualization
```

这里将 profiling 中的 topsys.stp 拷贝到 /tmp 目录，然后查看这个文件。

这段脚本的作用是每隔 5 秒钟列出当前系统中最高的 20 个 systemcalls。这个脚本对一些系统软件的开发人员作用较大。脚本如下：

```
#!/usr/bin/stap
#
# This script continuously lists the top 20 systemcalls in the interval
# 5 seconds
#
```

```

global syscalls_count

probe syscall.* {
    syscalls_count[name] <<< 1
}

function print_systop () {
    printf ("%25s %10s\n", "SYSCALL", "COUNT")
    foreach (syscall in syscalls_count- limit 20) {
        printf ("%25s %10d\n", syscall, @count(syscalls_count[syscall]))
    }
    delete syscalls_count
}

probe timer.s(5) {
    print_systop ()
    printf("-----\n")
}

```

下面使用 -v 参数试跑这个脚本，当输出为 pass 5 : starting run 时，就证明这个脚本通过了调试，可以编译成模块了。

```

[root@systemtap tmp]# stap -v topsys.stp
Pass 1: parsed user script and 103 library script(s) using 201636virt/29536res/3
156shr/26856data kb, in 120usr/10sys/142real ms.
Pass 2: analyzed script: 429 probe(s), 44 function(s), 41 embed(s), 1 global(s)
using 306308virt/135184res/4204shr/131528data kb, in 960usr/230sys/1863real ms.
Pass 3: translated to C into "/tmp/stap5s2AT2/stap_6421353660f490975a2a346ed9
c7ed0f_182267_src.c" using 306308virt/135576res/4596shr/131528data kb, in
40usr/30sys/79real ms.
Pass 4: compiled C into "stap_6421353660f490975a2a346ed9c7ed0f_182267.ko" in
6980usr/760sys/9346real ms.
Pass 5: starting run.

```

SYSCALL	COUNT
read	27
ppoll	25
fcntl	4
pselect6	1

通过测试之后，使用 -p4 -m <模块名> <脚本> 的命令行即可将脚本编译成一个模块，如下：

```

[root@systemtap tmp]# stap -v -p4 -m topsys.ko topsys.stp
Truncating module name to 'topsys'
Pass 1: parsed user script and 103 library script(s) using 201408virt/29580res/3
200shr/26628data kb, in 120usr/10sys/129real ms.
Pass 2: analyzed script: 429 probe(s), 44 function(s), 41 embed(s), 1 global(s)
using 306284virt/135196res/4224shr/131504data kb, in 1010usr/80sys/1090real ms.
Pass 3: translated to C into "/tmp/stapX0bYk3/topsys_src.c" using 306284virt/135

```

```

516res/4544shr/131504data kb, in 40usr/40sys/77real ms.
topsys.ko
Pass 4: compiled C into "topsys.ko" in 3900usr/180sys/4322real ms.
[root@systemtap tmp]# ll topsys.ko
-rw-r--r-- 1 root root 657770 Jun  1 00:34 topsys.ko

```

5. 测试模块

测试模块时，先在另外一台机器上安装 systemtap-runtime 包，然后使用 staprun 直接 run 编译好的模块。此时就可以看到有多少的 syscall 了。

下面用 cat 命令向 /dev/null 重定向写入，同时打开新的终端，运行 staprun topsys.ko 去查看 syscall 的实时情况。

```

[root@test ~]# staprun topsys.ko
      SYSCALL      COUNT
      read         27
      ppoll        25
      fcntl        4
      pselect6     1
-----

[root@test ~]# cat /dev/zero > /dev/null

[root@test ~]# staprun topsys.ko
      SYSCALL      COUNT
      read         1914489
      write        1914462
      ppoll        25
      fcntl        4
      pselect6     1
      poll         1
-----

      SYSCALL      COUNT
      read         1912028
      write        1912003
      ppoll        25
      rt_sigprocmask 4
      select       2
-----

```

可以看到，此时系统出现了大量的 read 和 write 的 syscall，两者相差数量不大，这也是比较典型文件复制时会出现的场景。

2.4 与内存相关的那些事情

2.4.1 内存泄漏

如果程序在运行过程中不能正常回收不用的内存，那么时间一长就会导致内存增长很

高,最终导致系统不可用,这种情况叫做内存泄漏。内存泄漏是一个让人烦恼的问题,不仅系统管理员烦恼,程序员也烦恼这个问题,所以定位分析内存泄漏是每一个系统管理员需要掌握的技能。

开源的 `valgrind` 是一个非常易于上手的内存分析工具,它可以分析内存泄漏、缓存命中等。本节以笔者工作中曾遇到的一个问题为例,讲述如何使用 `valgrind` 来定位内存泄漏问题。

笔者在生产环境使用 `Puppet` 作为自动化的基础工具,但是发现在 `CentOS 5` 的系统中,`Puppet` 进程一段时间之后使用了太多的系统内存,导致服务器宕机。根据监控系统可以看到 `Puppet` 进程的内存存在缓慢持续增长的趋势,因此怀疑 `Puppet` 有内存泄漏的问题,于是开始查找证据。

首先,用 `valgrind` 来分析 `Puppet` 在运行过程中是否出现内存泄漏问题。在 `leak summary` 中,很明确地指出了内存泄漏,泄漏了多少的量,如下:

```
[root@server1 ~]# valgrind --tool=memcheck /usr/sbin/puppetd -t
==1794== Memcheck, a memory error detector
==1794== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==1794== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==1794== Command: /usr/sbin/puppetd -t
==1794==
==1794== Conditional jump or move depends on uninitialised value(s)
.....
==1794==
==1794== HEAP SUMMARY:
==1794==    in use at exit: 23,430,539 bytes in 136,643 blocks
==1794==    total heap usage: 702,272 allocs, 565,629 frees, 262,185,929 bytes
    allocated
==1794==
==1794== LEAK SUMMARY:
==1794==    definitely lost: 617 bytes in 18 blocks
==1794==    indirectly lost: 0 bytes in 0 blocks
==1794==    possibly lost: 84,736 bytes in 1,523 blocks
==1794==    still reachable: 23,345,186 bytes in 135,102 blocks
==1794==    suppressed: 0 bytes in 0 blocks
==1794== Rerun with --leak-check=full to see details of leaked memory
==1794==
==1794== For counts of detected and suppressed errors, rerun with: -v
==1794== Use --track-origins=yes to see where uninitialised values come from
==1794== ERROR SUMMARY: 583211 errors from 100 contexts (suppressed: 19 from 6)
```

因为 `Puppet` 是由 `Ruby` 所写,所以内存泄漏的原因可能是来自 `Ruby` 本身。查看系统,得知安装的 `Ruby` 版本为 `1.8.5` 版本。在将 `Ruby` 升级到 `1.8.7` 版本之后,发现内存泄漏情况缓解了很多,不过依然有泄漏情况存在。示例如下:

```
[root@server1 ~]# rpm -qa | grep ruby
ruby-libs-1.8.5-31.el5_9
```

```
ruby-shadow-1.4.1-8.el5
ruby-1.8.5-31.el5_9
ruby-augeas-0.4.1-2.el5
```

可以看到,此时尽管内存泄露的情况缓解了很多,但是无法阻止问题再次发生,于是设置了一个 cronjob 定期重启 Puppet 进程,以保证避免内存泄漏而导致服务器宕机的问题发生。

2.4.2 虚拟内存、物理内存与页缺失

众所周知,在计算机发展的开始,内存是一个稀缺资源,即便现在的服务器动辄 128GB 的内存,它依然是一个稀缺的系统资源。所以内存管理也是影响性能的因素之一。

对于内存,首先,要清楚内存的单位, Paging 是内存的最小单位,称为页,类似磁盘的 block 概念,默认情况一个页是 4KB 大小。

通常情况下,对于内存的分配,并不是进程申请多少内存,操作系统就给多少内存。一般来说,当进程向操作系统申请 10GB 的内存时,操作系统收到请求后会进行自我检查,经过分析决定给予进程 10GB 的内存空间,此时操作系统会对这个进程说:“Hi,我可以给你 10GB 的内存空间。”这样的内存称为进程虚拟内存。而此时,操作系统并没有真正给予进程 10GB 内存,操作系统还会对进程说:“Hi,虽然我可以给你 10GB 内存空间,但现在你实际只需要 300MB 的内存就可以运行程序了,所以物理内存中现在只划分了 300MB 给你。”这种实际分配给进程的内存叫做物理内存。

在系统中可以使用 ps 查看进程的虚拟内存与物理内存大小,如下:

```
[root@server ~]# ps aux | head -1
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

其中, VSZ 这列代表的是虚拟内存, RSS 这列代表的是物理内存。

由于进程不能直接获取物理内存,而是每一个进程都有一个虚拟内存空间,所以虚拟内存不会直接映射到物理内存,直到使用的时候才会出现映射关系。这里就会产生一个问题,当进程向操作系统请求内存时,可能操作系统并没有准备好。这种情况叫做内存页缺失,页缺失有两种情况:一种是主页缺失,一种是次页缺失。

(1) 主页缺失

如果进程请求的数据不在物理内存中,要从磁盘或者交换分区换到内存中,这种叫做主页缺失,这种情况是非常影响性能的。

(2) 次页缺失

当第一次物理内存被使用的时候,物理内存中其实没有分配,这时候就产生了一个次页缺失。次页缺失对性能的影响不会特别大,一般情况下可以忽略。

下面说明如何查看进程的页缺失情况,笔者截取了生产系统中一台 KVM 主机的情况,如下:

```
[root@kvm01 ~]# ps o pid,comm,minflt,majflt `pidof qemu-kvm`
  PID COMMAND          MINFLT MAJFLT
 2834 qemu-kvm          443408224 117369
 2948 qemu-kvm          224733987 63896
19651 qemu-kvm          270213039 66725
20011 qemu-kvm          123041546 66718
20862 qemu-kvm          550133562 197954
26869 qemu-kvm          632148562 165397
32601 qemu-kvm          1103935202 165473
```

其中, MINFLT 是次页缺失, MAJFLT 是主页缺失。可以看到虚拟机比较繁忙, 有大量的主页和次页缺失。

2.4.3 Out of Memory

Out of Memory (OOM) 是系统管理员的另外一个噩梦。OOM 发生的原因主要在于, 当发生次页缺失的时候, 恰好系统无法再释放出物理内存, 此时系统只有杀掉一些进程来释放物理内存。

OOM 会选择占用内存最多的那个进程开始杀进程, 一直到内存足够为止。但是这样的方式往往会造成一些困扰, 因为关键进程被 kill 掉, 相当于宕机。其实可以让 OOM 不杀进程, 而是让 kernel panic, 可通过如下方式来实现。

设置 `/proc/sys/vm/panic_on_oom` 为 1, 这样在发生 OOM 的时候就会让 kernel panic, 示例如下:

```
[root@kvm01 ~]# cat /proc/sys/vm/panic_on_oom
0
[root@kvm01 ~]# echo 1 > /proc/sys/vm/panic_on_oom
```

实际情况中, 如果 OOM killer 发生了, 说明内存真的不足了。这时需要重视内存的使用情况, 评估是否需要增加内存。

2.4.4 Overcommit

前面说到了虚拟内存与物理内存的关系, 一般情况下进程并不会一次用光申请的内存, 所以操作系统为了提高内存使用率, 会向进程“超卖”内存, 以便能响应更多的进程内存申请, 当然, 操作系统有时候也是有点节操的, 它并不会无限制响应进程的内存申请, 这可防止内存申请过多, 导致操作系统本身的内存空间不足而无法正常运行。Linux 系统中使用 Overcommit 的方式来控制内存的申请。

控制是否允许内存“超卖”是通过 `/proc/sys/vm/overcommit_memory` 来实现的, 它有三种模式, 分别是 0、1、2。示例如下:

```
[root@kvm01 ~]# cat /proc/sys/vm/overcommit_memory
0
```

- ❑ 0 是系统默认的模式, 在这种模式下, 系统会尽可能地响应进程的内存申请, 这样一来, 有可能发生前面一节所说的 Out of Memory 情况。
- ❑ 1 的模式下, 系统完全响应进程的内存申请, 不管自己的资源还剩下多少。
- ❑ 2 的模式下, 系统完全不允许进程申请超过系统设置大小的内存空间。在这种模式下, 系统设置的可申请内存空间大小是:

$$\text{Swap} + \text{RAM} * (\text{" /proc/sys/vm/overcommit_ratio" } / 100)$$

其中, /proc/sys/vm/overcommit_ratio 就是系统最大可分配内存的百分比, 默认为 50, 也就是说最大为 50%。

在使用 2 模式时, 在 /proc/meminfo 中 CommitLimit 和 Committed_AS 这两个值会显得比较重要, CommitLimit 就是系统可以申请虚拟内存的最大值, Committed_AS 是系统已经申请的虚拟内存的大小。

2.4.5 cache 与 buffer

cache 与 buffer 这两个词都有缓存的意思, 故而成为大家争论的焦点, 而且在数据库中也有这两个词, 所以网络上众说纷纭, 这里只说在 free 命令中出现的 cache 与 buffer。

buffer 指的是索引信息, 就是对磁盘文件的索引缓存, 这个值一般比较低。

cache 则是传统意义上的缓存, 它缓存的是文件内容。

关于 buffer 和 cache 使用的计算方式请查看前面 2.2.2 节中 free 的用法。这里也给出一个示例, 如下:

```
[root@ash0007 ~]# free -m
```

	total	used	free	shared	buffers	cached
Mem:	23985	22409	1576	0	14	352
-/+ buffers/cache:	22041	1944				
Swap:	20294	1942	18352			

2.5 与磁盘相关的那些事情

2.5.1 HDD 与 SSD

硬件的发展真的非常快, 从 SSD 固态硬盘出现到大规模进入各个公司的生产环境仅仅几年时间, 同时 HDD 机械硬盘的发展也未停滞, HDD 硬盘虽然读写速度没有质的突破, 但是容量却在不断攀升, 现在市场上已经可以买到 8TB 的单块硬盘, 所以在未来的很长的一段时间里, HDD 和 SSD 会在服务器市场上并存。

(1) HDD

HDD 性能瓶颈主要两个方面: seek time 寻址时间与 rotational delay 旋转延时。

HDD 磁盘读数据的过程是先找到磁道, 然后找磁道上的扇区。找到磁道的时间叫做寻址时间, 也就是 seek time; 找到扇区的时间叫做旋转延时, 也就是 rotational delay, 这两

个动作的时间加起来可能会达到 1s 以上。所以针对 seek time 和 rotation delay 做优化时主要是减少寻找的时间，以及寻找的次数。

(2) SSD

SSD 发展迅速，变化也较快，接口包括 SATA、mSATA、PCI-E 等，存储的颗粒包括 MLC、SLC、TLC 等。

- ❑ MLC：一个存储单元存储多个（通常是两个）比特位的信息，寿命比 SLC 短，读写速度慢于 SLC，但是价格便宜。
- ❑ SLC：一个存储单元只存储一个比特位的信息，寿命长，读写速度快，价格昂贵。
- ❑ TLC：一个存储单元存储多个（通常是三个）比特位的信息，速度慢，寿命短，但是价格便宜，多用于手机存储芯片。

2.5.2 HDD 磁盘的调度算法

HDD 的速度较慢（这不是绝对的！），针对机械硬盘的特性，解决该问题有两种方式：

- 1) 加入中间缓存，比如增大 HDD 上的缓存。
- 2) 对于 I/O 请求合并操作，尽可能顺序写入，顺序读取。

根据这两种方式，操作系统会针对不同的应用场景采用不同的磁盘调度方式，查看系统配置的磁盘调度算法如下：

```
[root@server ~]# cat /sys/block/sda/queue/scheduler
noop anticipatory [deadline] cfq
```

系统提供了四种调度算法，CentOS 6 默认设置在 deadline 中，下面讲讲这四种调度算法的区别。

- ❑ noop（无操作等待算法）：不干预任何的 I/O 请求，直接将 I/O 请求交给存储设备，由存储设备自己完成。这个常出现在使用 SAN 的场景下，由存储自己完成 I/O 合并优化，或者出现在虚拟机上。宿主机自己会完成 I/O 请求的合并，虚拟机不需要做 I/O 请求合并。
- ❑ anticipatory（预期算法）：预期调度会将 I/O 请求放进队列，但并不立刻完成，而是在合并成顺序 I/O 后再完成请求，对于持续大量顺序 I/O 的场景适合使用 anticipatory。
- ❑ deadline（最后期限）：将 I/O 请求放进队列不处理，一直等到队列中的 I/O 请求多到足够合并成一个比较好的 I/O 请求为止。这个方式适合虚拟化环境的物理机，数据库服务器。
- ❑ cfq（完全公平队列）：对每一个进程的 I/O 请求公平处理，I/O 响应很快，适合随机存取，比如文件服务器。

如果需要更改磁盘的调度算法，只需用 echo 方式将算法写入即可：

```
[root@server ~]# echo cfq > /sys/block/sda/queue/scheduler
[root@server ~]# cat /sys/block/sda/queue/scheduler
```

```
noop anticipatory deadline [cfq]
```

2.5.3 文件系统上的日志

操作系统中数据写入磁盘的方式是先写入缓存，然后再写入磁盘。如果在数据写入了缓存，还没来得及没有写入磁盘的时候，机器断电了或者宕机了，那么在机器重新启动时就会发现实际数据和预期状态不一致。为了能恢复到一致，文件系统会从磁盘划分一个日志空间，在操作系统写数据到磁盘上之前，会将脏数据优先写入日志空间，然后再同步到磁盘。

文件系统的日志写入方式有以下三种。

- ❑ **ordered 方式**：只记录元数据到日志空间，待元数据写入日志空间之后，再把数据写入磁盘文件系统。这种方式下文件系统的性能和数据的安全性可以做到相对的均衡。这也是大多数日志文件系统默认的方式。
- ❑ **writeback 方式**：元数据和数据会同时写入磁盘，这种方式提供了较好的磁盘性能，但是数据安全性无法保证。
- ❑ **journal 方式**：这种方式会先向日志空间写入元数据和数据，然后向文件系统再写一次元数据和数据，这种方式数据最为安全，但是因为元数据和数据都会写两份，文件系统的性能也是最差的。

2.6 系统资源限制

系统资源是有限的，有的时候为了系统安全或者为了能承载更多的压力，需要限制或放开一些进程的资源使用。在早期的 Linux 系统中，一般用 `ulimit` 来限制进程的资源使用，在现在的 Linux 系统中，`kernel` 又引入了 `cgroup` 进一步加强限制进程的资源的使用。

2.6.1 ulimit

`ulimit` 几乎可以说是所有系统管理员都必须熟练掌握的一个配置，大部分的配置都在 `/etc/security/limits.conf` 中，配置文件中包含了绝大部分配置的解释。比如，如何限制用户进程数、如何确定打开文件数目等，这已经是大家很熟悉的配置了，这里不多讲解，下面用 `ulimit` 来演示如何限制内存申请。

前面的章节讲到了虚拟内存，虚拟内存是进程向操作系统申请的内存。虚拟内存的申请也是可以用 `ulimit` 来限制的。

首先，使用 `ulimit -a` 查看现在系统中 `ulimit` 的设置情况，如下：

```
[root@systemtap ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
```

```

scheduling priority      (-e) 0
file size                (blocks, -f) unlimited
pending signals          (-i) 14720
max locked memory        (kbytes, -l) 64
max memory size          (kbytes, -m) unlimited
open files               (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority       (-r) 0
stack size               (kbytes, -s) 10240
cpu time                 (seconds, -t) unlimited
max user processes       (-u) 14720
virtual memory          (kbytes, -v) unlimited
file locks               (-x) unlimited

```

可以看到, virtual memory 一行是 unlimited, 即无限的。

此时将 virtual memory 设置成 0, 看看会发生什么。

```

[root@systemtap ~]# ulimit -v 0
[root@systemtap ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority      (-e) 0
file size               (blocks, -f) unlimited
pending signals          (-i) 14720
max locked memory        (kbytes, -l) 64
max memory size          (kbytes, -m) unlimited
open files              (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority       (-r) 0
stack size               (kbytes, -s) 10240
cpu time                 (seconds, -t) unlimited
max user processes       (-u) 14720
virtual memory          (kbytes, -v) 0
file locks               (-x) unlimited

```

在执行 ls 命令的时候可以发现不会正确执行, 而是提示 Killed! 甚至连 reboot 都无法正确执行。如下:

```

[root@systemtap ~]# ls
Killed
[root@systemtap ~]# ls
Killed
[root@systemtap ~]# reboot
Killed

```

这是因为系统虚拟内存为 0, ls、reboot 无法申请到虚拟内存空间, 故而这个命令无法执行, 只能被系统杀掉!

2.6.2 Cgroup

ulimit 限制资源的方式显得比较粗旷，也无法限制磁盘 I/O，所以从 kernel 2.6.24 开始，引入了一个新的资源限制的方式——Cgroup。

Cgroup 中控制资源的系统成为 controller，Cgroup 提供了如下的 controller 来控制 CPU、内存、块设备、进程资源等。

- ❑ CPU/cpuacct/cpuset
- ❑ memory
- ❑ blkio
- ❑ device
- ❑ freezer
- ❑ net_cls

当 Cgroup 进程启动时，它会在系统中产生一个挂载点，在这个挂载点里含有 Cgroup 一切的配置，这些配置可以通过 echo <value> 的方式直接修改，也可以通过编译配置文件 /etc/cgconfig.conf 和 /etc/cgrules.conf 来让配置持久化。

Cgroup 的配置选项也颇多，下面使用实战的方式来理解 Cgroup 的原理，并且了解如何使用 Cgroup。

1. 安装 Cgroup

安装 libcgroup 包，启动 cgconfig 服务。会发现根目录中出现了 /cgroup 这个目录，同时里面还有一些子目录，如下：

```
[root@systemtap ~]# yum -y install libcgroup
[root@systemtap ~]# /etc/init.d/cgconfig start
[root@systemtap ~]# ls /cgroup/
blkio  cpu  cpuacct  cpuset  devices  freezer  memory  net_cls
```

2. 限制 Apache 内存使用

Cgroup 有两个主要的配置文件：/etc/cgconfig.conf 和 /etc/cgrules.conf，cgconfig.conf 用来定义资源限制的规则，比如可以使用多少内存、磁盘 I/O 等，cgrules.conf 用来定义哪些程序，或者哪些用户使用哪一种限制规则的。这里以限制 Apache 内存为例讲解基础配置。

在 /etc/cgconfig.conf 中，默认有一个 mount 规则，mount 规则的意义在于默认情况下需要对哪些资源做限制，比如不需要对磁盘 I/O 做限制，就可以去掉 blkio 这行。示例如下：

```
mount {
    cpuset    = /cgroup/cpuset;
    cpu       = /cgroup/cpu;
    cpuacct   = /cgroup/cpuacct;
    memory    = /cgroup/memory;
```

```

devices = /cgroup/devices;
freezer = /cgroup/freezer;
net_cls = /cgroup/net_cls;
blkio   = /cgroup/blkio;
}

```

规则的语法是 `group <name> { <controller> {<param name> = <param value>;} }`，这里要限制 Apache 可以使用的内存为 1MB，因此可在 `/etc/cgconfig.conf` 里写上如下代码：

```

group httpd {
    memory {
        memory.limit_in_bytes=102400;
        memory.swappiness=0;
    }
}

```

在 `cgrules.conf` 中加入如下一行，语义为所有用户执行 `apachectl` 这个命令时应用 `cgroup memory` 这个 controller，规则是 `cgconfig.conf` 中的 `httpd`。

```
*/usr/sbin/apachectl memory httpd
```

然后，重启服务让配置生效。

```

[root@systemtap ~]# /etc/init.d/cgconfig restart
[root@systemtap ~]# /etc/init.d/cgred restart

```

理论上来说，1MB 内存是无法运行 Apache 服务的，我们看看 Apache 究竟能不能启动。

使用 `apachectl` 命令启动 Apache 的时候，就发现这个进程被 kill 掉了，因为 Apache 申请的虚拟内存超过了 1MB 的大小，触发了系统的 OOM！

```

[root@systemtap ~]# apachectl restart
Killed

```

在日志中，可以很明确地看到 `apachectl` 被 kill 掉的全部行为。

```

Jun 14 15:56:12 systemtap kernel: apachectl invoked oom-killer: gfp_mask=0xd0,
order=0, oom_adj=0, oom_score_adj=0
Jun 14 15:56:12 systemtap kernel: [<ffffffff81127782>] ? oom_kill_process+0x82/
0x2a0
Jun 14 15:56:12 systemtap kernel: Task in /httpd killed as a result of limit of
/httpd
Jun 14 15:56:12 systemtap kernel: Memory cgroup out of memory: Kill process 2070
(apachectl) score 1000 or sacrifice child
Jun 14 15:56:12 systemtap kernel: Killed process 2070, UID 0, (apachectl) total-
vm:106072kB, anon-rss:116kB, file-rss:640kB

```

尝试将内存扩大到 10MB，看看 Apache 能不能启动。

将 `memory.limit` 的值设置为 10 485 760，然后重启 `cgconfig` 服务。

```

group httpd {
    memory {

```

```
memory.limit_in_bytes=10485760;
memory.swappiness=0;
}
}
```

此时可以看到 Apache 服务启动了，80 端口在正常的监听中。

```
[root@systemtap ~]# /etc/init.d/cgconfig restart
Stopping cgconfig service: [ OK ]
Starting cgconfig service: [ OK ]
[root@systemtap ~]# apachectl start
[root@systemtap ~]# netstat -nltp | grep 80
tcp        0      0 :::80                :::*                  LISTEN      2110/httpd
```

现在使用 ps 命令查看在 cgroup 中 httpd 进程的状态，可以看到 httpd 进程应用到了 memory 这个 controller 上。

```
[root@systemtap ~]# ps -eo pid,cgroup,cmd | grep httpd
2110 blkio://net_cls://freezer://devices://memory:/httpd;cpuacct://cpu://cpuset://usr/sbin/httpd -k start
2111 blkio://net_cls://freezer://devices://memory:/httpd;cpuacct://cpu://cpuset://usr/sbin/httpd -k start
2112 blkio://net_cls://freezer://devices://memory:/httpd;cpuacct://cpu://cpuset://usr/sbin/httpd -k start
2113 blkio://net_cls://freezer://devices://memory:/httpd;cpuacct://cpu://cpuset://usr/sbin/httpd -k start
2114 blkio://net_cls://freezer://devices://memory:/httpd;cpuacct://cpu://cpuset://usr/sbin/httpd -k start
2115 blkio://net_cls://freezer://devices://memory:/httpd;cpuacct://cpu://cpuset://usr/sbin/httpd -k start
2116 blkio://net_cls://freezer://devices://memory:/httpd;cpuacct://cpu://cpuset://usr/sbin/httpd -k start
2117 blkio://net_cls://freezer://devices://memory:/httpd;cpuacct://cpu://cpuset://usr/sbin/httpd -k start
2118 blkio://net_cls://freezer://devices://memory:/httpd;cpuacct://cpu://cpuset://usr/sbin/httpd -k start
2122 blkio://net_cls://freezer://devices://memory://cpuacct://cpu://cpuset://usr/sbin/httpd -k start
grep httpd
```

3. 限制磁盘 I/O

在现实环境中，可能需要对一些写入优先级不高的进程做 I/O 限制，比如一些级别很低的文件备份时，这里用 dd 命令来演示如何限制磁盘读写。

首先在 /etc/cgconfig.conf 中添加如下规则：

```
group diskio {
    blkio {
        blkio.throttle.read_bps_device="8:0 1048576";
        blkio.throttle.write_bps_device="8:0 20480";
    }
}
```

在 diskio 这个规则中, 使用 blkio.throttle.read/write_bps_device 来控制磁盘 I/O 速率, 这里还可以使用 blkio.throttle.read_iops_device 来控制 IOPS, 以达到同样的目的。

blkio.throttle.read/write_bps_device 后面的参数分别是磁盘号和限制的值。8: 0 是 /dev/sda 这个块设备号, 用 ls -l 命令查看, 可知 1 048 576 是最大读取速率, 这里则是 1MB。

```
[root@systemtap ~]# ll /dev/sda
brw-rw---- 1 root disk 8, 0 Jun 14 21:04 /dev/sda
```

然后在 /etc/cgrules.conf 中添加一个应用规则:

```
*:/bin/dd      blkio    diskio
```

一切配置完成, 重启 cgconfig 和 cgred 两个服务, 准备开始测试。

首先, 测试磁盘读 I/O 的限制。打开两个终端, 一个终端中开启 iotop, 另外一个终端中运行如下命令:

```
[root@systemtap ~]# dd if=/dev/sda of=/dev/null
```

此时可在 iotop 上看到, DISK READ 被精确地限制在了 1000KB 左右, 测试成功。

```
Total DISK READ: 1001.87 K/s | Total DISK WRITE: 0.00 B/s
```

TID	PRIO	USER	DISK READ	DISK WRITE	SWAPIN	IO>	COMMAND
1664	be/4	root	1001.87 K/s	0.00 B/s	0.00 %	97.97 %	dd if=/dev/sda of=/dev/null
1	be/4	root	0.00 B/s	0.00 B/s	0.00 %	0.00 %	init

终止 dd 命令之后, 也可以看到 dd 命令显示出每秒只能读取 1MB。

```
[root@systemtap ~]# dd if=/dev/sda of=/dev/null
^C252473+0 records in
252472+0 records out
129265664 bytes (129 MB) copied, 123.968 s, 1.0 MB/s
```

成功测试了磁盘读 I/O 之后, 再来测试磁盘写 I/O。因为 CentOS 6 内核中的 blkio 只支持磁盘 I/O 的 sync 和 direct 两种方式, 不支持 buffer 方式。所以这里将使用 dd 的 direct I/O 方式来测试。因为 direct I/O 速率较慢, 所以这里将写速率限制到 20KB, 以便能看到明显的效果。

运行如下命令:

```
[root@systemtap ~]# dd if=/dev/zero of=/tmp/file oflag=direct
```

从 iotop 上, 几乎可以看到 DISK WRITE 被控制在 20KB, 说明采用的方式成功了。

```
Total DISK READ: 0.00 B/s | Total DISK WRITE: 21.67 K/s
```

TID	PRIO	USER	DISK READ	DISK WRITE	SWAPIN	IO>	COMMAND
1702	be/4	root	0.00 B/s	21.67 K/s	0.00 %	99.99 %	dd if=/dev/zero of=/tmp/file oflag=direct

终止 dd 命令之后, 发现平均写约为 10KB, 低于设置的 20K。主要原因在于磁盘写 I/O 机制比读 I/O 要复杂很多, 在 2.6.32 这个内核中 blkio 还不能非常精确地控制好写 I/O。

```
[root@systemtap ~]# dd if=/dev/zero of=/tmp/file oflag=direct
^C577+0 records in
577+0 records out
295424 bytes (295 kB) copied, 27.5842 s, 10.7 kB/s
```

4. 限制虚拟机 CPU

虚拟机的 VCPU 一般来说是由 libvirtd 自动分配的，但是有时候为了减少虚拟化环境中的 CPU 切换，会将 VCPU 绑定在某一个物理 CPU 的核上。

首先，查看虚拟机 VCPU 现在的信息，如下：

```
[root@kvm ~]# virsh vcpuinfo guest
VCPU:      0
CPU:       5
State:     running
CPU time:  19927.5s
CPU Affinity:  YYYYYYYYYYYYYYYY

VCPU:      1
CPU:       8
State:     running
CPU time:  17281.8s
CPU Affinity:  YYYYYYYYYYYYYYYY
```

可以看到，两个 VCPU 分别在物理 CPU 的第 5 个和第 8 个核心上。

此时，重启 cgconfig 服务，然后重启 libvirtd 服务及虚拟机，保证 cgconfig 载入了虚拟机配置。

```
[root@kvm ~]# /etc/init.d/cgconfig restart
Stopping cgconfig service: [ OK ]
Starting cgconfig service: [ OK ]
[root@kvm ~]# /etc/init.d/libvirtd restart
Stopping libvirtd daemon: [ OK ]
Starting libvirtd daemon: [ OK ]
```

```
[root@kvm ~]# virsh destroy guest
[root@kvm ~]# virsh start guest
```

此时会看到 Cgroup 目录中产生了虚拟机的相应配置文件：

```
[root@kvm ~]# ls -l /cgroup/cpuset/libvirt/qemu/guest/
total 0
--w--w--w- 1 root root 0 Jun 15 15:45 cgroup.event_control
-r--r--r-- 1 root root 0 Jun 15 15:45 cgroup.procs
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.cpu_exclusive
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.cpus
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.mem_exclusive
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.mem_hardwall
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.memory_migrate
-r--r--r-- 1 root root 0 Jun 15 15:45 cpuset.memory_pressure
```

```

-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.memory_spread_page
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.memory_spread_slab
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.mems
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.sched_load_balance
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.sched_relax_domain_level
drwxr-xr-x 2 root root 0 Jun 15 15:45 emulator
-rw-r--r-- 1 root root 0 Jun 15 15:45 notify_on_release
-rw-r--r-- 1 root root 0 Jun 15 15:45 tasks
drwxr-xr-x 2 root root 0 Jun 15 15:45 vcpu0
drwxr-xr-x 2 root root 0 Jun 15 15:45 vcpu1

```

这里的重点是 **vcpu0** 和 **vcpu1** 两个文件，可以看到里面的内容是 0~15，意思就是这两个 VCPU 可以绑定在物理 CPU 的 0~15 个核的任意一个上。

```

[root@kvm ~]# cat /cgroup/cpuset/libvirt/qemu/guest/vcpu0/cpuset.cpus
0-15
[root@kvm ~]# cat /cgroup/cpuset/libvirt/qemu/guest/vcpu1/cpuset.cpus
0-15

```

下面使用 **echo** 的方式向文件中直接写入想绑定的物理 CPU 的核心上。

```

[root@kvm ~]# echo 9 > /cgroup/cpuset/libvirt/qemu/guest/vcpu0/cpuset.cpus
[root@kvm ~]# cat /cgroup/cpuset/libvirt/qemu/guest/vcpu0/cpuset.cpus 9
[root@kvm ~]# echo 10 > /cgroup/cpuset/libvirt/qemu/guest/vcpu1/cpuset.cpus
[root@kvm ~]# cat /cgroup/cpuset/libvirt/qemu/guest/vcpu1/cpuset.cpus 10

```

此时就可以看到 VCPU 已经被绑定到相应的 CPU 上了，如下：

```

[root@kvm ~]# virsh vcpuinfo guest
VCPU:          0
CPU:           9
State:         running
CPU time:      35.5s
CPU Affinity:   -----y-----

VCPU:          1
CPU:          10
State:         running
CPU time:      12.6s
CPU Affinity:   -----y-----

```

这是使用 **echo** 方式直接使虚拟机生效的方式，如果希望系统重启之后依然生效，那必须要写入 **cgconfig.conf** 配置文件中，控制 CPU 的 controller 是 **cpuset**，整个配置的写法如下：

```

group libvirt {
    cpuset {
        cpuset.cpus=0-15;
        cpuset.mems=0;
    }
}

```

```
group libvirt/qemu {  
    cpuset {  
        cpuset.cpus=0-15;  
        cpuset.mems=0;  
    }  
}  
  
group libvirt/qemu/guest {  
    cpuset {  
        cpuset.cpus=8-13;  
    }  
}  
group libvirt/qemu/guest/vcpu0 {  
    cpuset {  
        cpuset.cpus=9;  
    }  
}  
  
group libvirt/qemu/guest/vcpu1 {  
    cpuset {  
        cpuset.cpus=10;  
    }  
}
```


用户集中认证

3.1 openLDAP 简介

第 1 章讲解了 Linux 下的用户管理，而单机上的账号管理的弊端是显而易见的：当需要管理的主机越来越多的时候，用户不得不在每个主机上将同样的账号和密码再设置一遍，而当相关的人员离职时，又不得不将对应的账号做一次全部删除——也许有更好的办法来解决这个问题，但是这时候没有比使用一套用户集中认证的系统更好的办法。本章将从什么是 LDAP 入手，讲清 LDAP 的原理，演示 Linux 下常用 openLDAP 的安装配置。

什么是 LDAP

LDAP 是 lightweight directory access protocol 的缩写，即轻量级目录访问协议，用于快速查询用户记录。LDAP 使用树状的结构来存取记录，“树”又称为 OU，最上面的一层（根部）叫做“基准 DN”，DN 可以理解为用户，而用户还可以有更小的用户，但是根据协议，LDAP 最大分为 4 层，读者可以把 LDAP 想象成一颗倒置的树。

3.2 openLDAP 的安装

顾名思义，openLDAP 是 LDAP 的开源实现，目前默认安装在众多流行的 Linux 发行版中，openLDAP 主要包含 4 个部分：

- ❑ 独立 LDAP 守护进程 slapd
- ❑ 独立 LDAP 更新复制进程 slurpd
- ❑ LDAP 协议库

❑ 工具软件和客户端

它的安装非常简单，只需一条命令即可：

```
[root@localhost ~]# yum install openldap-*
Is this ok [y/N]: y
Downloading Packages:
.....(省略安装输出)
Complete!
```

3.3 openLDAP 的配置

准备 LDAP 相关配置文件：

```
cd /etc/openldap/
cp /usr/share/openldap-servers/slapd.conf.obsolete?slapd.conf
cp /usr/share/openldap-servers/DB_CONFIG.example /var/lib/ldap/DB_CONFIG
```

创建 LDAP 管理密码：

```
[root@localhost openldap]# slappasswd
New password: #这里输入的是openldap，下同，读者请自行修改。
Re-enter new password:
{SSHA}1xw6C6/1Rm7lmOu8Gmi4fq1xrTrTgyyq
```

打开 slapd.conf，找到 rootpw，并作相关修改：

```
# Cleartext passwords, especially for the rootdn, should
# be avoided. See slappasswd(8) and slapd.conf(5) for details.
# Use of strong authentication encouraged.
# rootpw                secret
rootpw                  {SSHA}1xw6C6/1Rm7lmOu8Gmi4fq1xrTrTgyyq
```

测试生成配置文件：

```
[root@localhost openldap]# slaptest -f slapd.conf -F /etc/openldap/slapd.d/
config file testing succeeded #看到这个输出，就是成功的
```

启动 slapd 服务：

```
[root@localhost ~]# chkconfig slapd on
[root@localhost ~]# /etc/init.d/slapd start
Starting slapd: [ OK ]
[root@localhost ~]# netstat -lntp | grep 389
tcp  0  0  0.0.0.0:389      0.0.0.0:*        LISTEN      1288/slapd
tcp  0  0  :::389         :::*             LISTEN      1288/slapd
```

至此，openLDAP 的安装就完成了，但是 openLDAP 并不能直接读取系统用户信息（openLDAP 需要从自己的数据文件中读取用户数据），所以这时需要安装 migrationtools，用这个工具协助将系统用户导出为 openLDAP 可以读取的文件，示例如下：

```
[root@localhost openldap]# yum install migrationtools
```

修改 `/usr/share/migrationtools/migrate_common.ph`，找到如下两行配置，并对其值进行相应的修改，如下所示：

```
# Default DNS domain
$DEFAULT_MAIL_DOMAIN = "my-domain.com";

# Default base
$DEFAULT_BASE = "dc=my-domain,dc=com";
```

添加一个用户用于测试：

```
[root@localhost migrationtools]# useradd ldaptest
[root@localhost migrationtools]# passwd ldaptest
Changing password for user ldaptest.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

将系统用户导出为 `ldif` 文件，并将生成的文件导入到 `openLDAP` 数据库中：

```
[root@localhost migrationtools]# cd /usr/share/migrationtools
./migrate_base.pl > /tmp/base.ldif
./migrate_passwd.pl /etc/passwd > /tmp/passwd.ldif
./migrate_group.pl /etc/group > /tmp/group.ldif
```

#使用 `ldapadd` 添加记录时，这里输入的密码是 `openldap`，读者可根据自己实际设置的密码调整。

```
[root@localhost ~]# ldapadd -D "cn=Manager,dc=my-domain,dc=com" -W -f /tmp/base.ldif
```

```
Enter LDAP Password:
adding new entry "dc=my-domain,dc=com"
adding new entry "ou=Hosts,dc=my-domain,dc=com"
adding new entry "ou=Rpc,dc=my-domain,dc=com"
adding new entry "ou=Services,dc=my-domain,dc=com"
adding new entry "nisMapName=netgroup.byuser,dc=my-domain,dc=com"
adding new entry "ou=Mounts,dc=my-domain,dc=com"
adding new entry "ou=Networks,dc=my-domain,dc=com"
adding new entry "ou=People,dc=my-domain,dc=com"
adding new entry "ou=Group,dc=my-domain,dc=com"
adding new entry "ou=Netgroup,dc=my-domain,dc=com"
adding new entry "ou=Protocols,dc=my-domain,dc=com"
adding new entry "ou=Aliases,dc=my-domain,dc=com"
adding new entry "nisMapName=netgroup.byhost,dc=my-domain,dc=com"
```

```
[root@localhost ~]# ldapadd -D "cn=Manager,dc=my-domain,dc=com" -W -f /tmp/
passwd.ldif
```

```
Enter LDAP Password:
adding new entry "uid=root,ou=People,dc=my-domain,dc=com"
adding new entry "uid=bin,ou=People,dc=my-domain,dc=com"
adding new entry "uid=daemon,ou=People,dc=my-domain,dc=com"
adding new entry "uid=adm,ou=People,dc=my-domain,dc=com"
```

```

adding new entry "uid=lp,ou=People,dc=my-domain,dc=com"
adding new entry "uid=sync,ou=People,dc=my-domain,dc=com"
adding new entry "uid=shutdown,ou=People,dc=my-domain,dc=com"
adding new entry "uid=halt,ou=People,dc=my-domain,dc=com"
adding new entry "uid=mail,ou=People,dc=my-domain,dc=com"
adding new entry "uid=uucp,ou=People,dc=my-domain,dc=com"
adding new entry "uid=operator,ou=People,dc=my-domain,dc=com"
adding new entry "uid=games,ou=People,dc=my-domain,dc=com"
adding new entry "uid=gopher,ou=People,dc=my-domain,dc=com"
adding new entry "uid=ftp,ou=People,dc=my-domain,dc=com"
adding new entry "uid=nobody,ou=People,dc=my-domain,dc=com"
adding new entry "uid=vcsa,ou=People,dc=my-domain,dc=com"
adding new entry "uid=saslauth,ou=People,dc=my-domain,dc=com"
adding new entry "uid=postfix,ou=People,dc=my-domain,dc=com"
adding new entry "uid=sshd,ou=People,dc=my-domain,dc=com"
adding new entry "uid=ldap,ou=People,dc=my-domain,dc=com"
adding new entry "uid=ldaptest,ou=People,dc=my-domain,dc=com"

```

```

[root@localhost migrationtools]# ldapadd -D "cn=Manager,dc=my-domain,dc=com" -W
-f /tmp/group.ldif

```

Enter LDAP Password:

```

adding new entry "cn=root,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=bin,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=daemon,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=sys,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=adm,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=tty,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=disk,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=lp,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=mem,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=kmem,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=wheel,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=mail,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=uucp,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=man,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=games,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=gopher,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=video,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=dip,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=ftp,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=lock,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=audio,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=nobody,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=users,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=utmp,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=utempter,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=floppy,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=vcsa,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=cdrom,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=tape,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=dialout,ou=Group,dc=my-domain,dc=com"

```

```

adding new entry "cn=saslauth,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=postdrop,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=postfix,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=sshd,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=ldap,ou=Group,dc=my-domain,dc=com"
adding new entry "cn=ldaptest,ou=Group,dc=my-domain,dc=com"

```

最后重启 `sldapd`，服务端的配置就完成了。

```
[root@localhost migrationtools]# /etc/init.d/slapd restart
```

3.4 利用 openLDAP 集中认证

在客户机上安装 `client` 工具：

```
[root@localhost ~]# yum install nss-pam-ldapd pam_ldap openldap-clients authconfig
sssd-*
```

编辑 `/etc/openldap/ldap.conf`：

```

BASE      dc=my-domain,dc=com
URI       ldap://192.168.188.128 #openLDAP服务器的IP地址

```

编辑 `/etc/nsswitch.conf`：

```

passwd:    files ldap
shadow:    files ldap
group:     files ldap

```

运行以下命令，这会配置 `/etc/sss/sss.conf`、`/etc/sysconfig/authconfig`，并启动 `sssd` 服务：

```

[root@localhost ~]# authconfig --enableldap --enableldapauth --ldapserver=
192.168.188.128 --ldapbasedn="ou=People,dc=my-domain,dc=com" --enablemk-
homedir --update
Starting sssd: [ OK ]

```

编辑 `/etc/pam.d/system-auth`，添加相关配置，见粗体字部分：

```

#%PAM-1.0
# This file is auto-generated.
# User changes will be destroyed the next time authconfig is run.
auth      required      pam_env.so
auth      sufficient    pam_unix.so try_first_pass nullok
auth      sufficient    pam_ldap.so
auth      required      pam_deny.so

account   required      pam_unix.so
account   [default=bad success=ok user_unknown=ignore] pam_ldap.so

password  requisite     pam_cracklib.so try_first_pass retry=3 type=

```

```

password      sufficient      pam_unix.so try_first_pass use_authtok nullok sha512
shadow
password      sufficient      pam_ldap.souseauthtok
password      required        pam_deny.so

session        optional      pam_keyinit.so revoke
session        required      pam_limits.so
session        [success=1 default=ignore] pam_succeed_if.so service in crond quiet
use_uid
session        required      pam_unix.so
session        optional      pam_ldap.so

```

最后重启 sshd 服务：

```

[root@localhost ~]# service sshd restart
Stopping sshd: [ OK ]
Starting sshd: [ OK ]

```

至此客户端的配置就完成了，只需退出当前登录，然后尝试使用之前创建的 ldaptest 用户登录到系统中。



域名服务器 DNS

4.1 DNS 服务简介

每台联网服务器都需要一个 IP 地址唯一标识，在互联网成立之初，全球的计算机数量屈指可数，所以为了方便互相通信，当时人们使用类似于早期的电话号码本来记录每台主机的 IP 地址——这在当时是可以接受的。然而，随着时代的发展，主机的数量越来越多，这时候再使用这样的方法就行不通了，于是人们发明了一种更为方便的方式，这就是 DNS 服务。拿我们日常生活中访问某个网站为例，我们在浏览器的地址栏中输入 `www.xyz.com`，浏览器其实并不知道这个域名对应的主机 IP 是什么，所以它必须首先解析出该域名的 IP 地址——通过向域名服务器请求解析；域名服务器回应给浏览器对应的 IP 后，浏览器才能真正地向该域名发起请求。

一个典型的域名是由顶级域名、二级域名和主机名构成。例如 `www.zzz.com`，它的顶级域名是 `com`，二级域名是 `zzz`，主机名是 `www`。

如图 4-1 所示，域名服务至少由根域（也是一个点）、顶级域（常见的 `com`、`net`、`org` 等都在此列）、二级域、主机名组成。所以域名系统是一个分层的树形结构。

当主机需要解析某个域名时，它会向其配置的域名服务器发起查询，依然以使用浏览器访问 `www.zzz.com` 为例：首先浏览器会检查其自身是否有缓存（假设之前访问过这个域名，那么在该域名最大缓存时间过期之前，浏览器可以记住这个域名对应的 IP），如果缓存中没有查询到或是缓存已过期，浏览器则将检查本地（具体便是 `/etc/hosts` 文件）是否存有该记录，如果没有就向配置的 DNS 服务器发起域名解析请求，接到请求的 DNS 服务器首先也会查询自己的缓存，如果也没有则检查客户端想要查询的域名是否自己可以解析，如

果不能解析, 则将会向其他 DNS 服务器发起请求——这是一个递归查询的过程。

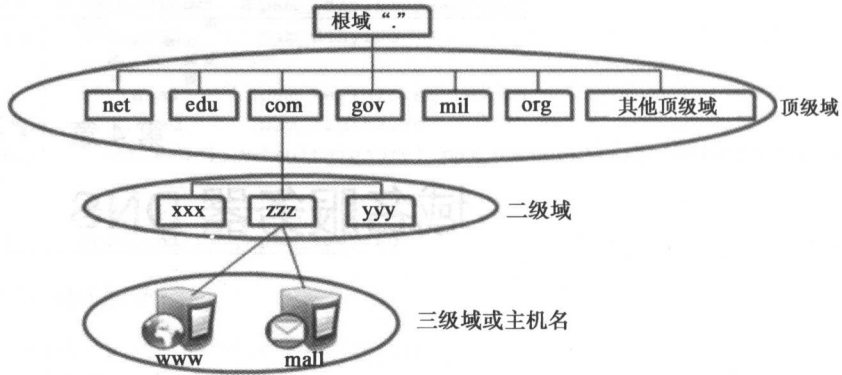


图 4-1 域名服务结构

4.2 DNS 安装配置

4.2.1 DNS 安装过程

DNS 的安装过程比较简单, 方式如下:

```
[root@localhost ~]# yum install bind bind-chroot bind-utils
Loaded plugins: fastestmirror
Setting up Install Process
Loading mirror speeds from cached hostfile
* base: mirrors.pubyun.com
* extras: mirrors.pubyun.com
* updates: centos.ustc.edu.cn
Package 32:bind-utils-9.8.2-0.37.rc1.el6_7.5.x86_64 already installed and latest
version
Resolving Dependencies
--> Running transaction check
---> Package bind.x86_64 32:9.8.2-0.37.rc1.el6_7.5 will be installed
---> Package bind-chroot.x86_64 32:9.8.2-0.37.rc1.el6_7.5 will be installed
--> Finished Dependency Resolution
```

Dependencies Resolved

Package	Arch	Version	Repository	Size
Installing:				
bind	x86_64	32:9.8.2-0.37.rc1.el6_7.5	updates	4.0 M
bind-chroot	x86_64	32:9.8.2-0.37.rc1.el6_7.5	updates	74 k

Transaction Summary

```
=====
Install          2 Package(s)
```

```
Total download size: 4.1 M
```

```
Installed size: 7.3 M
```

```
Is this ok [y/N]:y
```

4.2.2 关于 chroot 的解释

有必要解释一下为什么需要安装 bind-chroot 这个包：DNS 服务会在操作系统运行后由系统用户 root 启动，如果 DNS 存在漏洞就可能会被黑客加以利用。而 bind-chroot 会将 bind 进程严格限制在特定的目录中，一旦进程试图脱离该目录，就会立即失去所有权限，所以如果黑客试图通过 DNS 进程的漏洞攻击系统，即便获得了某些权限，也只能在有限范围内破坏。安装了 bind-chroot 后，DNS 的配置将使用 /var/named/chroot 目录，配置文件放置于 /var/named/chroot/etc 中，数据文件放在 /var/named/chroot/var/named 中。在 CentOS 5 以及更老的版本中，所有的配置文件和数据文件都必须在 /var/named/chroot 中，在 CentOS 6 之后，可以直接使用默认的目录和配置，但是 DNS 是在 chroot 环境中运行的——关于这点，感兴趣的读者可以研读一下 /etc/init.d/named 启动脚本。

4.2.3 配置主配置文件

按如下方式修改两处主配置文件 /etc/named.conf，保存退出即可。

```
options {
    listen-on port 53 { any; }; #修改为any
    listen-on-v6 port 53 { ::1; };
    directory      "/var/named";
    dump-file       "/var/named/data/cache_dump.db";
    statistics-file  "/var/named/data/named_stats.txt";
    memstatistics-file "/var/named/data/named_mem_stats.txt";
    allow-query     { any; }; #修改为any
    recursion yes;

    dnssec-enable yes;
    dnssec-validation yes;
    dnssec-lookaside auto;

    /* Path to ISC DLV key */
    bindkeys-file "/etc/named.iscdlv.key";

    managed-keys-directory "/var/named/dynamic";
};

logging {
    channel default_debug {
        file "data/named.run";
```

```

        severity dynamic;
    };

};

zone "." IN {
    type hint;
    file "named.ca";
};

include "/etc/named.rfc1912.zones";
include "/etc/named.root.key";

```

4.2.4 DNS 的正向解析配置

所谓正向解析，即查询域名所对应的 IP，正向解析配置写在 `/etc/named.rfc1912.zones` 中。这里假设要解析一个域 `test.com`，则修改该配置文件为：

```

[root@localhost ~]# cat /etc/named.rfc1912.zones
// named.rfc1912.zones:
//
// Provided by Red Hat caching-nameserver package
//
// ISC BIND named zone configuration for zones recommended by
// RFC 1912 section 4.1 : localhost TLDs and address zones
// and http://www.ietf.org/internet-drafts/draft-ietf-dnsop-default-local-
// zones-02.txt
// (c)2007 R W Franks
//
// See /usr/share/doc/bind*/sample/ for example named configuration files.
//

zone "test.com" IN {
    type master;
    file "test.com.zone";
};

```

该配置文件的意思是，要查询 `test.com` 的 IP 记录，请参照文件 `test.com.zone`，所以还需准备好这个配置文件。具体方法如下：

```

cd /var/named/
[root@localhost named]# cp named.localhost test.com.zone
[root@localhost named]# chown root:named test.com.zone

```

然后针对 `test.com.zone` 做如下修改：

```

[root@localhost named]# cat test.com.zone
$TTL 1D
@      IN SOA  test.com.      admin.test.com. (
                                0      ; serial
                                1D      ; refresh
                                1H      ; retry

```

```

                                1W      ; expire
                                3H )    ; minimum
@      IN      NS      dns.test.com.

dns     IN      A       10.50.63.185
www     IN      A       10.50.63.185

```

该文件的意思是，www.test.com 域名和 dns.test.com 的 IP 应该是 10.50.63.185，读者可根据实际情形自行修改。

启动 named，并使用 dig 命令测试，可以看到 dig 命令查询到的结果和配置的 IP 是一致的，说明配置生效了。

```

[root@localhost named]# /etc/init.d/named start
Starting named:                                     [ OK ]

[root@localhost named]# dig @localhost www.test.com

; <<>> DiG 9.8.2rc1-RedHat-9.8.2-0.37.rc1.el6_7.5 <<>> @localhost www.test.com
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 63123
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 1

;; QUESTION SECTION:
;www.test.com.                IN      A

;; ANSWER SECTION:
www.test.com.                 86400   IN      A      10.50.63.185

;; AUTHORITY SECTION:
test.com.                     86400   IN      NS      dns.test.com.

;; ADDITIONAL SECTION:
dns.test.com.                 86400   IN      A      10.50.63.185

;; Query time: 0 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Jan 17 19:10:40 2016
;; MSG SIZE rcvd: 80

```

请注意配置文件的所有者必须是 named，否则可能出现的情况是：服务重启成功但解析不到，同样 DNS 反解也是如此。

4.2.5 DNS 的反向解析配置

通过域名查找 IP 的方式称为正向解析，反之，通过 IP 查询其对应的域名则称为反向解析。这里可能有读者会有疑问：如果说 DNS 正解是可以理解的，那么 DNS 反解到底是

为什么呢？作为一个用户来说，是不太可能在意 IP 到底对应什么域名的。实际上，DNS 反解更多的是为了满足某些应用的需求，特别是邮件服务。在垃圾邮件当道的时代，邮件服务器在收到邮件发送请求之前，会反向解析一下其 IP 地址的域名，如果该域名不在其允许列表之内，则会拒绝该请求。所以说，DNS 反解更多的是应用程序从安全角度考虑的需求。

编辑 `/etc/named.rfc1912.zones`，在最后添加以下内容：

```
zone "63.50.10.in-addr.arpa" IN {
    type master;
    file "10.50.63.zone";
    allow-update { none; };
};
```

这里需要注意的是，zone `"63.50.10.in-addr.arpa"` 是 IP 地址的反写，假设需要反解的区域 IP 地址是 A.B.C.D，则 zone 应该写为：`"C.B.A.in-addr.arpa"`。

编辑文件 `10.50.63.zone`，如下所示：

```
[root@i-lgc0hf6i named]# cat 10.50.63.zone
$TTL 1D
@      IN SOA  test.com.      admin.test.com. (
                        0      ; serial
                        1D     ; refresh
                        1H     ; retry
                        1W     ; expire
                        3H )   ; minimum
@      IN     NS      dns.test.com.

185    IN     PTR     dns.test.com. #第一列的数字为该主机IP末位
185    IN     PTR     www.test.com.
```

重启服务，并进行反向解析测试，可以看到 ANSWER SECTION 部分反解正确。

```
[root@i-lgc0hf6i named]# /etc/init.d/named restart
Stopping named: [ OK ]
Starting named: [ OK ]
[root@i-lgc0hf6i named]# dig @localhost -x 10.50.63.185

; <<>> DiG 9.8.2rc1-RedHat-9.8.2-0.37.rc1.el6_7.5 <<>> @localhost -x 10.50.63.185
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 15408
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 1, ADDITIONAL: 1

;; QUESTION SECTION:
;185.63.50.10.in-addr.arpa.      IN      PTR

;; ANSWER SECTION:
185.63.50.10.in-addr.arpa. 86400 IN      PTR      www.test.com.
185.63.50.10.in-addr.arpa. 86400 IN      PTR      dns.test.com.
```

```
;; AUTHORITY SECTION:
63.50.10.in-addr.arpa. 86400 IN NS dns.test.com.

;; ADDITIONAL SECTION:
dns.test.com. 86400 IN A 10.50.63.185

;; Query time: 0 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sat Feb 6 16:06:24 2016
;; MSG SIZE rcvd: 117
```

4.2.6 利用 DNS 实现负载均衡

以 Web 应用为例,在公司发展初期,也许一台服务器就已经足够,但随着业务的发展,单台服务器的负载越来越高,平行地添加更多服务器就成为最简单的扩容方法。当 DNS 中配置的一个域名对应多个 IP 时, DNS 将会依次返回客户端不同的 IP,这样就达到了负载均衡的效果。

将 test.com.zone 中 www 的记录添加多条,重启 named 服务后再多次使用 dig 测试,可以看到 DNS 每次都会返回三条记录,每次返回记录的第一条将会轮询出现,而每次浏览器只会使用第一条记录的 IP 去访问该域名。示例代码如下:

```
[root@localhost named]# cat test.com.zone
$TTL 1D
@ IN SOA test.com. admin.test.com. (
                                0 ; serial
                                1D ; refresh
                                1H ; retry
                                1W ; expire
                                3H ) ; minimum
@ IN NS dns.test.com.

dns IN A 10.50.63.185
www IN A 10.50.63.185
www IN A 10.50.63.186
www IN A 10.50.63.187

[root@i-lgc0hf6i ~]# /etc/init.d/named restart
Stopping named: . [ OK ]
Starting named: [ OK ]

[root@i-lgc0hf6i ~]# dig @localhost www.test.com

;<<>> DiG 9.8.2rc1-RedHat-9.8.2-0.37.rc1.el6_7.5 <<>> @localhost www.test.com
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17273
```

```
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 1, ADDITIONAL: 1

;; QUESTION SECTION:
;www.test.com.                IN      A

;; ANSWER SECTION:
www.test.com.                86400   IN      A      10.50.63.185
www.test.com.                86400   IN      A      10.50.63.186
www.test.com.                86400   IN      A      10.50.63.187

;; AUTHORITY SECTION:
test.com.                    86400   IN      NS      dns.test.com.

;; ADDITIONAL SECTION:
dns.test.com.                86400   IN      A      10.50.63.185

;; Query time: 0 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sat Feb  6 18:03:02 2016
;; MSG SIZE rcvd: 112
```

4.3 DNS 的主从复制

使用 DNS 的主从复制功能可以实现的功能非常明显：

- ❑ 提供冗余，避免单点故障；
- ❑ 均衡负载查询需求，从而提高系统可用性。

本节将演示 DNS 主从复制的配置方式，假设另一台 DNS 服务器的 IP 地址为：10.50.82.44，首先同样请按照 4.2.1 节中的方式安装相关软件包，此处不再演示。

在主服务器上，修改配置如下：

```
[root@i-lgc0hf6i ~]# cat /etc/named.rfc1912.zones
// named.rfc1912.zones:
//
// Provided by Red Hat caching-nameserver package
//
// ISC BIND named zone configuration for zones recommended by
// RFC 1912 section 4.1 : localhost TLDs and address zones
// and http://www.ietf.org/internet-drafts/draft-ietf-dnsop-default-local-
zones-02.txt
// (c)2007 R W Franks
//
// See /usr/share/doc/bind*/sample/ for example named configuration files.
//

zone "test.com" IN {
    type master;
    file "test.com.zone";
```

```

allow-update { none; };
allow-transfer { 10.50.82.44; };
};

zone "63.50.10.in-addr.arpa" IN {
    type master;
    file "10.50.63.zone";
    allow-update { none; };
    allow-transfer { 10.50.82.44; };
};

```

在从 DNS 上，配置文件如下：

```

[root@i-caccdm ~]# cat /etc/named.rfc1912.zones
// named.rfc1912.zones:
//
// Provided by Red Hat caching-nameserver package
//
// ISC BIND named zone configuration for zones recommended by
// RFC 1912 section 4.1 : localhost TLDs and address zones
// and http://www.ietf.org/internet-drafts/draft-ietf-dnsop-default-local-
zones-02.txt
// (c)2007 R W Franks
//
// See /usr/share/doc/bind*/sample/ for example named configuration files.
//

zone "test.com" IN {
    type slave;
    file "slaves/test.com.zone";
    masters { 10.50.63.185; };
};

zone "63.50.10.in-addr.arpa" IN {
    type slave;
    file "slaves/10.50.63.zone";
    masters { 10.50.63.185; };
};

```

注意从 DNS 在启动之前，/var/named/slaves/ 目录下为空，但是只需启动服务，该目录下的文件立刻从主 DNS 上同步了。查看这两个文件的内容，若和主 DNS 是一致的，说明同步成功了。

```

[root@i-caccdm ~]# ll /var/named/slaves/
total 0
[root@i-caccdm ~]# /etc/init.d/named start
Generating /etc/rndc.key: [ OK ]
Starting named: [ OK ]
[root@i-caccdm ~]# ll /var/named/slaves/
total 8

```



```
-rw-r--r-- 1 named named 345 Feb  7 13:32 10.50.63.zone
-rw-r--r-- 1 named named 353 Feb  7 13:32 test.com.zone
```

在从 DNS 上使用 dig 命令进行测试, 查询结果一致, 说明主从同步配置成功了。

```
[root@i-caccdm ~]# dig @localhost -x 10.50.63.185

; <<>> DiG 9.8.2rc1-RedHat-9.8.2-0.37.rc1.el6_7.6 <<>> @localhost -x 10.50.63.185
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 39884
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 1, ADDITIONAL: 1

;; QUESTION SECTION:
;185.63.50.10.in-addr.arpa.      IN      PTR

;; ANSWER SECTION:
185.63.50.10.in-addr.arpa. 86400 IN      PTR      www.test.com.
185.63.50.10.in-addr.arpa. 86400 IN      PTR      dns.test.com.

;; AUTHORITY SECTION:
63.50.10.in-addr.arpa. 86400 IN      NS      dns.test.com.

;; ADDITIONAL SECTION:
dns.test.com.            86400 IN      A      10.50.63.185

;; Query time: 1 msec
;; SERVER: ::1#53(::1)
;; WHEN: Sun Feb  7 14:01:38 2016
;; MSG SIZE rcvd: 117
```

4.4 配置纯缓存的 DNS 服务

纯缓存的 DNS 服务器就是本身并不维护 zone 文件, 而只是简单地将 DNS 请求转发给制定的 DNS 服务, 并将返回结果再返回给客户端, 同时将该结果记录在系统缓存中, 等待下次有同样的请求时, 直接将该记录返回给客户端。纯缓存的 DNS 服务配置非常简单, 按下面的配置完成后, 重启 named 服务即可。

```
[root@i-lgc0hf6i ~]# cat /etc/named.conf
//
// named.conf
//
// Provided by Red Hat bind package to configure the ISC BIND named(8) DNS
// server as a caching only nameserver (as a localhost DNS resolver only).
//
// See /usr/share/doc/bind*/sample/ for example named configuration files.
//
```

```

options {
    listen-on port 53 { any; };
    directory      "/var/named";
    dump-file       "/var/named/data/cache_dump.db";
    statistics-file "/var/named/data/named_stats.txt";
    memstatistics-file "/var/named/data/named_mem_stats.txt";
    allow-query     { any; };
    recursion yes;
    forward only;
    forwarders { 8.8.8.8; }; #将DNS请求转发给谷歌域名服务器
};

logging {
    channel default_debug {
        file "data/named.run";
        severity dynamic;
    };
};

```

使用 dig 查询域名，并得到了返回，如下：

```

[root@i-lgc0hf6i ~]# dig @localhost www.baidu.com

;<<>> DiG 9.8.2rc1-RedHat-9.8.2-0.37.rc1.el6_7.6 <<>> @localhost www.baidu.com
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->HEADER<- opcode: QUERY, status: NOERROR, id: 42386
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.baidu.com.                IN      A

;; ANSWER SECTION:
www.baidu.com.                436     IN      CNAME   www.a.shifen.com.
www.a.shifen.com.            299     IN      A       220.181.111.188
www.a.shifen.com.            299     IN      A       220.181.112.244

;; Query time: 1488 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Feb  7 14:27:20 2016
;; MSG SIZE rcvd: 90

```

4.5 DNS 的客户端配置

4.5.1 Linux 中的配置

在 Linux 系统中，配置系统使用制定的 DNS 服务器只需要简单地修改 `/etc/resolv.conf`

(修改 nameserver 行) 即可, 例如想要使用 10.50.63.185 作为系统的 DNS 服务器, 只需将 /etc/resolv.conf 修改为以下内容即可:

```
[root@i-lgc0hf6i ~]# cat /etc/resolv.conf
# Generated by NetworkManager
nameserver 10.16.10.3
```

4.5.2 Windows 中的配置

以 Windows7 为例, 打开“控制面板\网络和 Internet\网络连接”, 在网卡上右键, 选择属性, 在“网络”标签中, 选择“Internet 协议版本 4”, 点击“属性”打开“常规”标签, 点选“使用下面的 DNS 服务器地址”, 输入 DNS 的 IP 地址后, 按“确定”即可。

系统备份

5.1 为什么要备份

在管理一个 Linux 集群的工作中，最为重要却又最容易被人忽视的一点就是备份。一些初级的系统管理员往往没有意识到，维护系统里的信息比维护计算机硬件资源更加重要，硬件资源可以重新购置，可信息数据一旦丢失，就轻易无法弥补了。与此同时，虽然大部分资深的系统管理员能够认识到备份的必要性，但是由于资源备份很不幸是比较乏味的一个任务，而且有的时候，备份处于一个尴尬的地位——假设备份完成了，但是没有系统故障，也没有发生数据丢失，那么备份的作用永远都无法体现，于是许多系统管理员就会想，暂时就这样吧，下次再做，从而简单地跳过了这一任务。

但是，在一个 Linux 集群的建设中，需要重点指出备份的重要性。数据丢失的方式有千万种，我们永远无法预测下一次事故发生的原因是什么。有可能是经典的硬件故障、软件 bug，也有可能是无法避免的人为错误，毕竟我们永远不能保证“rm -rf/”这样的事情不会发生，就像我们不能避免极端的自然灾害一样，它们有可能会对业务、对数据造成巨大损害。显然备份不可能完美地解决所有问题，但是当这些事情发生的时候，我们希望至少有一种方法或者可能性，将数据尽可能恢复到最近的某一个时间点的状态，从而使其对业务的影响降到最低。

在备份的时候，系统管理员还需要精心地选择备份什么内容，大而全的备份往往是不切实际的，而且会消耗不必要的资源。应该根据数据业务来决定备份的内容，若是发生了影响实际业务的数据丢失事故，备份的内容应该能够保证业务受最小的影响。因此对于一个管理员来说，了解备份应该做什么，比备份能做什么更加重要。

5.2 常见的备份机制

现在假设我们的 Linux 系统上有 /data 这个文件目录，这个目录包含了所有与当前业务有关的用户数据。现在要备份这个目录。

这里马上就有两个问题需要考虑：

- 1) 什么时候做备份？备份频率是多少，每天一次？每周一次？
- 2) 怎么备份？每次备份是将所有的数据都直接全部保存成一个副本，还是有其他更好的方法？

显然这个问题没有什么标准答案，这里简单地介绍一下不同的备份机制，对于不同的数据，选用正确的备份机制是非常重要的。

5.2.1 完全备份

完全备份（full backup）是指将所有需要备份的文件和目录全部备份起来的一种方法。完全备份一般是增量备份（incremental backup）和差异备份（differential backup）的基础。一次完全备份后面一般跟有多次增量或差异备份，之后会重新做一次完全备份。

比如我们认为 /data 目录必须每周的工作日都做完全备份，那么意味着从周一到周五，/data 下面所有的内容每天都会被重复拷贝一次，即使当天其内容没有任何改变，数据也会被重新备份。

完全备份的优点是：

- ❑ 所有的文件都被备份，管理方便，恢复迅速。
- ❑ 不同历史备份版本的管理也比较简单。

它的缺点是：

- ❑ 由于需要备份所有文件，创建备份所需要的时间比较长。
- ❑ 完全备份需要消耗的存储空间比较大。

5.2.2 增量备份

增量备份是指针对上次备份以来的所有更改而做备份。上一次的备份可以是完全备份，也可以是增量备份。一般来说增量备份会以一次完全备份开始，接下来只备份那些改变了或新增的文件。

比如我们的备份任务是每周工作日的增量备份，那么周一第一天应该做一次完全备份，因为此时还没有备份过任何文件，此后每天都只备份和前一天相比更改过或新增的文件。

增量备份的优点有：

- ❑ 备份的速度比完全备份快。
- ❑ 备份所需存储空间比完全备份、差异备份都小。

增量备份的缺点有：

- 文件恢复比完全备份和差异备份慢。
- 文件恢复的方法比较复杂，必须用到所有的增量备份。

以之前的工作日备份方案为例，如果我们要恢复周五的备份文件，那么需要基于周一的完全备份，将之后周一至周五的所有增量备份合并起来，最终得到最新的备份文件。这个过程比较复杂，也比较耗时。

5.2.3 差异备份

差异备份只备份从上一次完全备份到当前时间为止改变过或者增减过的文件。差异备份的前一次备份基础一定是一次完全备份。因此如果需要恢复文件，一份完全备份和一份差异备份就足够了。

比如，备份任务是每周工作日进行差异备份。那么周一我们需要创建一份完全备份，因为此时的文件数据还没有被备份过。此后的每一天，我们都基于这一份完全备份，备份从周一到当天改变过或者增减过的文件。也就是说周三的差异备份也会包括周二的差异备份，而对于增量备份来说，周三的备份不包括周二的差异备份。

差异备份的优点有：

- 差异备份速度比完全备份快。
- 需要的存储空间比完全备份少。
- 文件恢复速度比增量备份快。

差异备份的缺点有：

- 备份速度比增量备份慢。
- 备份所需存储空间比增量备份多。
- 文件恢复比增量备份简单，但是比完全备份复杂，因为我们需要在完全或者增量备份中定位文件，因此速度也比完全备份慢。

5.3 Bacula 简介

5.3.1 什么是 Bacula

创建数据备份的方法有许多种，直接将文件拷贝到其他地方并保存起来也可以称之为文件备份。显然这样的备份方案过于人工，我们需要通过一些工具将备份这件工作变得简单且自动化起来。

Bacula 就是这么一款优秀的备份软件。它是一款开源的、企业级的计算机备份开源软件，分别有社区版和企业版。Bacula 使用服务器/客户端模式，其服务端几乎可以运行在所有的类 Unix 操作系统上面，当然也包括 Linux 以及其对应的各种发行版，其客户端支持更加广泛的操作系统，包括 Linux、OS X、Windows、Unix 等。

Bacula 的特色为：

- ❑ 多平台支持。
- ❑ 有标准的服务器 / 客户端模式。
- ❑ 有可配置的服务器 / 客户端验证方法。
- ❑ 支持数据备份的加密。
- ❑ 支持数据备份的一致性验证。
- ❑ 有丰富的后端数据库支持, 包括 MySQL、SQLite、PostgreSQL。
- ❑ 有丰富的配置管理接口, 可以通过命令行、GUI 和 Web 接口来管理配置和备份。
- ❑ 备份工作前后可以自定义执行脚本或者命令。

5.3.2 Bacula 的基本组件

Bacula 由 5 个基本组件构成: 控制器 (Director)、控制台 (Console)、文件管理器 (File)、存储管理器 (Storage) 和监控平台。其基本的体系结构如图 5-1 所示。

Bacula 控制器是监管所有备份、恢复和验证工作的守护进程。系统管理员使用 Bacula 控制台向控制器安排备份和恢复工作。

Bacula 控制台是用户与 Bacula 控制器交互的终端, 它可以是命令行模式, 也可以是 GUI 的方式。用户可以在任何地方启动控制台, 不需要和控制器运行在同一台计算机上面。

Bacula 文件服务, 即 Bacula 的客户端, 是一个安装在被备份的机器上的守护进程。备份进行时, 它负责将客户机上文件的信息和数据发送给 Bacula 存储守护进程; 在恢复备份的时候, 它还负责将正确的备份文件写入正确的系统位置。

Bacula 存储守护进程负责管理备份的存储介质, 其主要工作是从存储上读取和写入正确的备份文件。

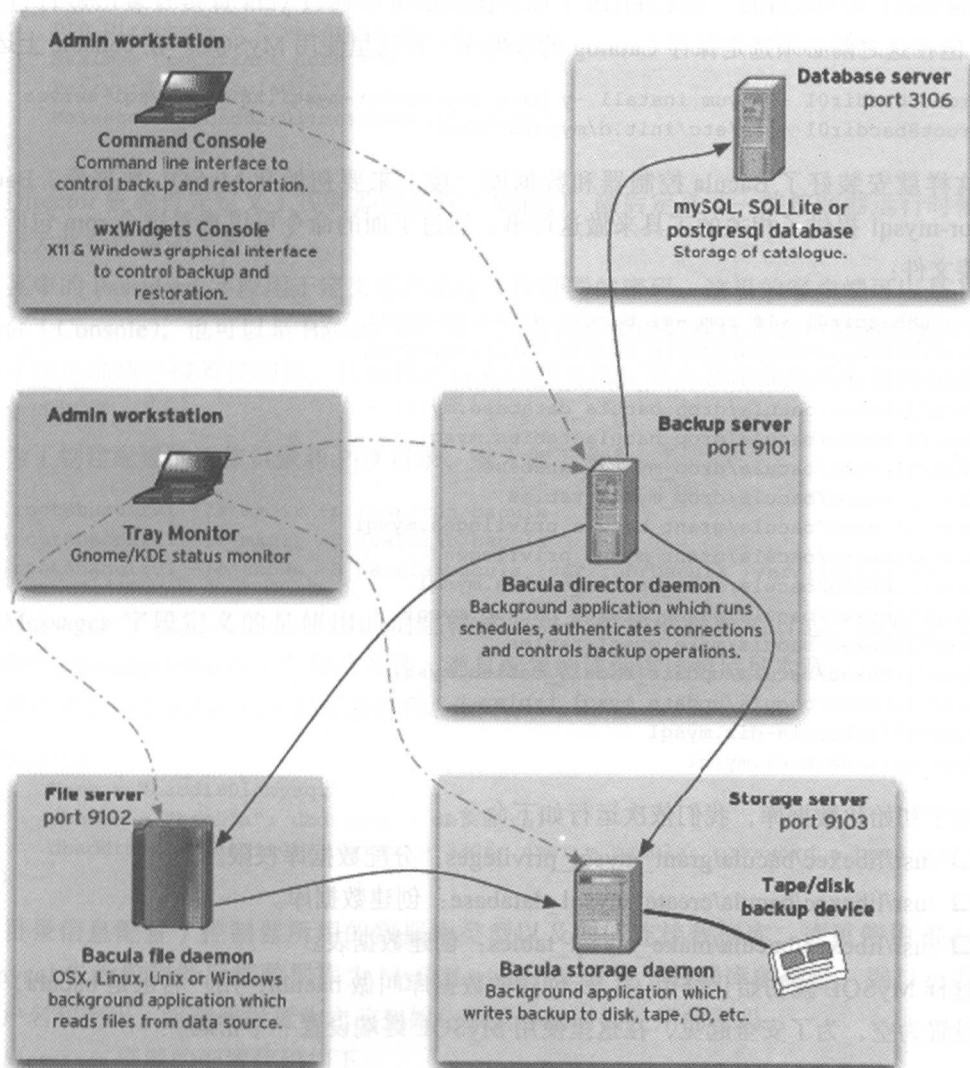
Bacula 目录 (catalog) 是 Bacula 控制器用来保存和维护备份任务和备份文件的服务。区别于使用基本的 tar、zip 来创建备份, Bacula 的一大特色就是能够对保存的文件和备份任务进行自动化管理。Catalog 使得系统管理员能够迅速地定位和恢复文件。Catalog 保存在控制器的后端数据库中, 当前支持 MySQL、PostgreSQL 以及 SQLite。

5.4 Bacula 的安装和配置

本章会详细地介绍 Bacula 相关各个组件的安装和配置。在介绍配置之前, 先来介绍一下 Bacula 系统中进行备份的一些基本概念。

- ❑ 作业 (Job) 和计划 (Schedule)。Bacula 将一次备份操作称为 Job。备份作业一般包含 FileSet、Client 和 Schedule。FileSet 可以理解为需要备份的文件集, Client 理解为需要备份的客户机, Schedule 表示备份计划, 也就是何时进行备份操作。
- ❑ Pool、Volume 和 Label。如果读者是第一次接触 Bacula, 可能会觉得这些概念比较难理解。Volume 是单一的一个备份介质, 比如一块磁盘, 或者一个文件, Bacula

会将数据写入到 Volume 中；一组 Volume 的集合称之为 Pool，Pool 保证了当一个 Volume 满了之后，Bacula 可以自动找到下一个 Volume，并写入备份数据；Volume 在被使用之前，必须被 Bacula 打标，从而保证被正确的读写。



Bacula application interactions

Note that these applications may actually run on fewer machines than shown here. You could run everything on one machine if you only wanted to back up a local disk to a local tape or disk.

Port numbers are the defaults and can be changed.

图 5-1 Bacula 的基本组件

5.4.1 Bacula 控制器

1. 安装 Bacula 控制器

下面来安装 Bacula 的控制器服务。在本书中，用到了两台 CentOS 6 的虚拟机，分别为 bacdir01 和 bacdir02。在 CentOS 下面，Bacula 可以通过 yum 直接安装已经打包好的 rpm。但在这之前必须选定保存 Catalog 的数据库，在这里使用 MySQL。在 console 上运行：

```
[root@bacdir01 ~]# yum install -y bacula-director-mysql.x86_64 mysql-server
[root@bacdir01 ~]# /etc/init.d/mysqld start
```

这样就安装好了 Bacula 控制器和数据库。接下来要初始化 MySQL 数据库。Bacula-director-mysql 提供了相关的工具来做这件事。通过下面的命令可以查看这个 rpm 包所包含的安装文件：

```
[root@bacdir01 ~]# rpm -ql bacula-director-mysql
/usr/libexec/bacula/create_bacula_database.mysql
/usr/libexec/bacula/create_mysql_database
/usr/libexec/bacula/drop_bacula_database.mysql
/usr/libexec/bacula/drop_bacula_tables.mysql
/usr/libexec/bacula/drop_mysql_database
/usr/libexec/bacula/drop_mysql_tables
/usr/libexec/bacula/grant_bacula_privileges.mysql
/usr/libexec/bacula/grant_mysql_privileges
/usr/libexec/bacula/make_bacula_tables.mysql
/usr/libexec/bacula/make_catalog_backup.mysql
/usr/libexec/bacula/make_mysql_tables
/usr/libexec/bacula/update_bacula_tables.mysql
/usr/libexec/bacula/update_mysql_tables
/usr/sbin/bacula-dir.mysql
/usr/sbin/dbcheck.mysql
```

为了初始化数据库，我们依次运行如下命令：

- ❑ /usr/libexec/bacula/grant_mysql_privileges：分配数据库权限。
- ❑ /usr/libexec/bacula/create_mysql_database：创建数据库。
- ❑ /usr/libexec/bacula/make_mysql_tables：创建数据表。

这样 MySQL 就初始化结束了，所创建的数据库叫做 bacula，用户名也是 bacula，密码初始设置为空，为了安全起见，在这里使用 MySQL 终端设置一个密码。

```
mysql> GRANT ALL PRIVILEGES ON `bacula`.* TO 'bacula'@'localhost' identified by
'baculadb';
Query OK, 0 rows affected (0.00 sec)
```

2. 配置 Bacula 控制器

Bacula 控制器的主要配置文件是 /etc/bacula/bacula-dir.conf。这是一个非常复杂的配置文件，由 Director、Catalog、Messages、Console、Storage、Pool、Counter、Fileset、Schedule、JobDefs、Job 等资源组成。

下面分别来介绍这些资源，首先要介绍的是 Director，其配置代码如下：

```
Director {
    Name = "bacdir01:director"
    Query File = "/etc/bacula/scripts/query.sql"
    Working Directory = "/var/lib/bacula"
    PID Directory = "/var/run/bacula"
    Maximum Concurrent Jobs = 5
    Password = "baculadir"
    Messages = "bacdir01:messages:standard"
}
```

Director 资源首先定义了控制器的名称 (Name)，随后定义了一些控制器运行时候的属性，最大并行作业的个数为 5。

其中的 Password 字段用于定义客户端连接控制器的密码。这里的客户端可以是 Bacula 控制台 (Console)，也可以是 Bacula 文件服务。密码必须配置在这些客户端的配置文件中，这样才能正确地连接到控制器。比如控制台安装在本机，那么就需要将其配置到控制台的配置文件里面。

为了创建配置的工作目录和 PID 目录，键入如下命令：

```
[root@bacdir01 ~]# mkdir -p /var/run/bacula
[root@bacdir01 ~]# mkdir -p /var/lib/bacula
[root@bacdir01 ~]# chown -R bacula:bacula /var/run/bacula /var/lib/bacula/
```

Messages 字段定义的是使用的消息格式。对于控制器参数的日志，使用定义好的“bacdir01:messages:standard”格式发送，消息配置的详细讲述稍后见本章。

然后来介绍 Catalog 相关的配置代码，如下：

```
Catalog {
    Name = "bacdir01:mysql"
    dbname = "bacula"; dbdriver = dbi:mysql
    dbaddress = bacdir01; dbport = 3306; user = bacula; password = baculadb
}
```

目录信息配置了控制器所用的数据库类型以及如何连接数据库，这里使用 dbdriver=dbi.mysql 告诉控制器后端数据库为 MySQL，接下来设置了数据库的地址、端口、连接所需用户名和密码，将密码设置为之前初始化数据库的时候使用的密码。

Messages 资源的配置代码如下：

```
Messages {
    Name = "bacdir01:messages:standard"
    Mail Command = "/usr/sbin/bsmtp -h localhost -f bacula@your.host.com -s
        \"Bacula Notice (from Director %d)\" %r"
    Mail = alerts@your.host.com = all, !skipped
    Console = all, !skipped, !saved
    Append = "/var/log/bacula/bacdir01:director.log" = all, !skipped
}
```

Messages 资源定义的是消息的格式以及发送消息的方法。在这里定义了上文所需要的“bacdir01:messages:standard”。Mail Command 配置发送邮件使用的系统命令，Mail Command 支持基本的宏操作，用来自定义发送邮件的标题和内容。比如 %d 表示 Bacula 控制台的名称、%r 表示收件人等。除此之外，还有一个配置参数 Operator Command，当 Bacula 的工作需要人工干预的时候，其消息用 Operator Command 发送。

接下来所配置的是消息发送的目的地，以及发送哪些消息。配置的一般格式有两种，如下：

```
destination = message-type1, message-type2, ...
destination = address = message-type1, message-type2, ...
```

这里的 destination 可以是 Mail，表示 Mail Command 的收件人地址；Operator，表示 Operator Command 的收件人地址；Console，表示 Bacula 控制台；File（文件）此文件会被覆盖；Append，表示追加到某个文件；Syslog，log 服务器；Catalog，表示发送到 Catalog 数据库等。

常见的 Message-type 包括：all，表示所有消息；skipped，表示文件在备份时被跳过的情况下产生的消息；saved，文件备份时产生的消息；fatal/error/warning/info，对应各种日志级别的消息。

消息的维护是一个值得关注的问题。在一个繁忙的备份系统中，产生的消息量可能非常大。如果将所有的消息都发送到 Console，Console 在启动的时候就会被消息刷屏；如果将消息都保存到 Catalog，则会对数据库的性能造成极大的影响。笔者推荐对消息进行有效的过滤，将不同消息发送到不同的目的地。

下面再来介绍 Storage 资源，配置代码如下：

```
Storage {
    Name = "bacsd01:storage:default"
    Address = bacsd01
    Password = "baculasd"
    Device = "FileStorage"
    Media Type = File
}
```

Storage 资源定义了 Bacula 控制器如何连接和使用存储守护进程。Address 字段配置存储守护进程的地址为 bacdir01，这是笔者内网的一台虚拟机；Password 字段配置了连接到 bacsd01 的密码。

其中的 Device 字段用于配置设备的名字，这个设备必须定义在存储守护进程的配置中。Media Type 可以是任意的值，表示存储的类型，一般建议设置成为有意义的字符串，比如 File，表示保存在磁盘文件中。

现在介绍 Pool 资源，配置代码如下：

```
Pool {
```

```

Name = "bacsd01:pool:default"
Label Format = "default.${JobId}.${Year}${Month:p/2/0/r}${Day:p/2/0/r}.
               ${Hour:p/2/0/r}${Minute:p/2/0/r}"
Pool Type = Backup
Recycle = No
Auto Prune = Yes
Volume Retention = 5 Week
# Don't allow re-use of volumes; one volume per job only
Maximum Volume Jobs = 1
}

```

Pool 定义了一组 Volume 的集合；Label 字段设置如何替 Volume 打标，这一选项一般在设置自动打标的时候用到，它 also 支持一些基本的变量替换，比如 \${JobID} 表示备份作业的 Id 号。

Volume Retention 表示 Catalog 中 Volume 信息的过期时间，待 Volume 过期后，如果设置了 Auto Prune，Bacula 会在 Catalog 数据库中自动将相关的数据清除，此时如果设置了 Recycle 为 Yes，这个 Volume 可以在下次被其他的备份作业使用。注意 Auto Prune 并不会自动清除备份文件，它只会清除在 Catalog 中的信息。Maximum Volume Jobs 表示最大的作业数量，这里设置为 1，表示一个 Volume 只保存一份备份文件，结合文件备份的方法，那么就表示一个备份文件中只包含一次备份数据。

FileSet 相关的配置代码如下：

```

FileSet {
    Name = "config:etc"
    Include {
        Options {
            Signature    = MD5
            Compression = GZIP
        }
        File = /etc
    }
    Exclude {
        File = /etc/puppet/modules
    }
}

```

FileSet 定义了需要备份的文件集。Name 表示此 FileSet 的名字，它可以被 Job 资源所引用；Include 首先定义了一系列备份的选项，使用 Gzip 进行压缩，同时为了保证安全性，使用 MD5 进行 hash 验证。接下来 File 选项设置了需要备份的文件或者目录，这个选项可以重复设置多次。Bacula 会递归备份目录，但是不会跨分区，因此如果一个目录下有多个分区的话，需要显式地将所有的挂载点配置进来。Exclude 表示在 Include 的目录中需要排除的文件和目录。

Schedule 资源相关的配置代码如下：

```

Schedule {
    Name = "Monthly:onMonday"
    Run = Level=Full First Mon at 16:30
    Run = Level=Differential Second-Fifth Mon at 16:30
    Run = Level=Incremental Tue-Sun at 18:40
}

```

Schedule 资源用来定义备份的时间和频率，以及备份的方式。用户可以定义多个 Schedule，用不同的名字来区分它们。比如这里定义了一个名为 Monthly:onMonday 的备份计划，每个月的第一个周一在 16:30 的时候做一次完全备份，第二个到第五个周一做一次差异备份，其他的日期里，每天在 18:40 的时候做一次增量备份。

Run 字段的语法为：

```
Run = Job-overrides Date-time-specification
```

其中，Job-overrides 为一系列的键值对，比如 Level=Full，用于定义一次全备；还可以设置 Pool、Storage、Messages 等，这些参数可以被 Job 作业中设置的参数覆盖。Date-time-specification 的语法比较复杂，一般需要设置月、日、时和分四个参数，比如 2nd-5th sun at 2:05，表示每个月的第二到第五个周日的 2 点 05 分，建议读者参阅相关文档了解 Date-time-specification 的详细写法。

Run 字段可以定义多次，所有定义的时段都会运行备份任务。如果两个 run 字段时间重复，备份作业会在同一时间运行两次。

Client 资源相关的配置代码如下：

```

Client {
    Name      = "bacsd01"
    Address   = "bacsd01"
    Password  = "baculafd"
    Catalog   = "bacdir01:mysql"
    File Retention = 6 Weeks
    Job Retention  = 3 Months
    Auto Prune = Yes
}

```

Client 资源用于定义控制器的客户端，也就是需要备份的机器。每一个客户机都必须在控制器的配置文件中被定义。

Address 字段定义了客户机的链接地址，可以是 DNS 主机名，也可以是 IP 地址；Password 字段配置的是连接客户机文件服务使用的密码，必须和客户机配置的密码一致；Catalog 定义了此客户端信息保存到的 Catalog 目录。

File Retention 字段用于设置 Catalog 保存客户端备份文件信息的时间。从备份作业结束时计算，如果时间超过了设置的 File Retention 值，且 AutoPrune 设置为 Yes，Bacula 会在 Catalog 数据库中将这些文件的信息删除。同理 Job Retention 设置的是备份作业保存的时间长短，如果超过了 Job Retention 配置值，相关备份作业在数据库中的信息将会被删除。

当一个备份的 Job 被删除时，其备份文件的相关信息也会在 Catalog 中被删除，因此一般建议 File Retention 的值小于 Job Retention 的值。注意 Bacula 并不会删除实际的备份文件，只是将数据库中关于这次备份的信息清除。

AutoPrune 为 Yes 时，Bacula 会在每次备份作业结束之后清除超过 Retention 的文件和作业信息。如果 AutoPrune 设置为 No，每一次作业结束后，保存在 Catalog 数据库中的信息都会增长。为了方便维护 Catalog，笔者建议在设置好 Retention 的前提下开启 AutoPrune。

下面轮到 Job 资源了，配置代码如下：

```
Job {
    Name = "bacsd01:default:etc"
    Type = Backup
    Schedule = "Monthly:onMonday"
    client = bacsd01
    FileSet = "config:etc"
    Storage = "bacsd01:storage:bacsd01:etc"
    Pool = "bacsd01:pool:default"
    Messages = "bacdir01:messages:standard"
    Full Max Run Time = 12 Hours
    Differential Max Run Time = 4 Hours
    Incremental Max Run Time = 2 Hours
}
```

Job 资源定义的是一次 Bacula 的作业，它可以是备份，也可以是恢复，由字段 Type 决定——Backup 或 Restore。这里配置了作业的客户机是 bacsd01，类型是备份作业，计划是之前定义的 Monthly:onMonday，Bacula 会按照定义好的时间自动执行备份任务。备份的文件集是前文中的 bacsd01:etc，使用标准消息机制，同时还配置好了 Storage 和 Pool。

对于一个作业，可以通过 Max Run Time 定义其运行的最长时间，超过这个时间的作业会被杀死。默认的 Max Run Time 是 6 个小时。同时对于不同的备份类型，也可以设置不同的时间，在这里我们对完全备份、差异备份和增量备份配置了不同的运行超时时间。

作业还可以配置在其运行之前执行系统命令，响应的配置选项有 Run Before Job、Run After Job、Run After Failed Job、Client Run Before Job、Client Run After Job 等。Run Before Job 表示在作业运行之前执行的命令，比如：

```
Run Before Job = "echo test"
```

如果命令返回值非 0，Bacula 会取消此备份作业。Run After Job 如果返回非 0，Bacula 会打印一条告警消息。Client Run Before Job 的不同之处是命令是在客户机上执行的。

为了执行备份恢复，还需要定义一个恢复作业：

```
Job {
    name = bacdir01:restore:default
    Enabled = No
}
```

```

Type = Restore
level = Incremental
Client = bacsd01
FileSet = "config:etc"
Storage = "bacsd01:storage:default"
Pool = "bacsd01:pool:default"
Messages = "bacdir01:messages:standard"
Where = '/tmp/restore'
}

```

Restore Job 里面的许多选项都没有实际作用，只是出于配置文件语法需要而写在里面。实际需要设置的参数有：**Where**，表示恢复到哪个目录；**Storage**，表示从哪个存储守护进程获得备份文件。数据的恢复作业需要手工唤起，它不会自动运行。

5.4.2 Bacula 存储守护进程

1. 安装 Bacula Storage

本书中安装存储守护进程的机器名叫做 bacsd01。CentOS 下可以用 yum 直接安装 Bacula Storage:

```
[root@bacdir01 ~]# yum install -y bacula-storage-common
```

2. 配置 Bacula Storage

存储守护进程的工作是：根据控制器的指令，接受从文件守护进程发送过来的数据并传递给存储进行备份；或者从存储中找到相关备份文件，发送给文件守护进程进行备份恢复。

Bacula 存储守护进程的默认配置文件是 /etc/bacula/bacula-sd.conf。相比于 bacula-dir.conf，这个文件的配置简单了许多，只需要配置 Storage、Director、Device 和 Message 这四种资源。

示例如下：

```

Director {
    Name = "bacdir01:director"
    Password = "baculadir"
}
Messages {
    Name = "bacsd01:messages:standard"
    Director = "bacdir01:director" = all
}
Storage {
    Name = "bacsd01:storage"
    Working Directory = "/var/lib/bacula"
    PID Directory = "/var/run/bacula"
    Maximum Concurrent Jobs = 32
}
Device {

```

```

Name = "FileStorage"
Media Type = File
Device Type = File
Archive Device = /opt/bacula/default
Label Media = Yes
Random Access = Yes
Automatic Mount = Yes
Removable Media = No
Always Open = No
}

```

存储守护进程必须配置唯一的 Storage 资源，其配置选项都很直观。Director 资源用来配置允许访问存储守护进程的控制器，Name 和 Password 字段必须和 bacula-dir.conf 中 Director 资源配置一致。Message 资源定义将消息发送到什么地方，这里配置将所有的消息都发送给控制器。

Device 资源配置存储守护进程使用的存储设备，存储设备可以配置多个。Device Type 用来配置设备的类型，可以是 File，表示文件设备，也可以是 Tape，表示磁带机，还可以是 FiFO。Archive Device 是必须配置的选项，对于 Tape 设备，这里应该设置设备的路径，如果是保存在磁盘中的文件，这里配置目录的绝对路径。

LabelMedia 选项表示自动给 Device 打标签，设备加入的 Pool 必须设置 Label 参数才能自动生成标签。对于磁带机，Random Access 必须设置为 no，其他情况下设置为 yes。

5.4.3 Bacula 客户端文件守护进程

将 File Daemon 安装在 bacsd01 上，作为控制器的一个客户机。在 CentOS 上可以通过 yum 直接安装，命令如下：

```
[root@bacsd01 ~]# yum install -y bacula-client
```

文件守护进程的配置文件是 /etc/bacula/bacula-fd.conf，它的配置相对比较简单，笔者使用的配置如下：

```

Director {
    Name = "bacdir01:director"
    Password = "baculafd"
}

FileDaemon {
    Name = "bacsd01"
    Working Directory = /var/lib/bacula
    PID Directory = /var/run/bacula
    Maximum Concurrent Jobs = 3
}

```

```

Messages {
    Name = "bacdir01:messages:standard"
}

```



```
Director = "bacdir01:director" = all, !skipped, !restored
}
```

Director 资源配置可以连接本客户端的控制器，可以配置多个实例，从而允许多个控制器连接客户端，Password 字段必须和控制器里 Client 资源配置的密码一致。FileDaemon 资源配置了客户端相关的属性。Message 资源配置消息发送到何处。

5.4.4 Bacula 控制台

Bacula 提供一个基于 Console 界面的控制台 bconsole，rpm 包为 bacula-console：

```
[root@bacdir01 ~]# yum install -y bacula-console
```

其配置文件是 /etc/bacula/bconsole.conf：

```
Director {
    Name = "bacdir01:monitor:director"
    Address = bacdir01.example.com
    Password = "baculadir"
}
```

这里只需配置控制器的相关信息，包括控制器服务器的地址和连接密码。

大多数系统管理员会发现使用 bconsole 来管理 Bacula 已经足够了，如果喜欢图形客户端，Bacula 还提供了 bat 这个 GUI 工具。在 CentOS 下面，它的包名字叫做 bacula-console-bat，读者可以自行使用。

5.4.5 启动服务

Bacula 的各个组件都配置好之后，分别在 bacdir01 上启动控制器，在 bacsd01 上启动存储服务和文件服务：

```
[root@bacdir01 ~]# /etc/init.d/bacula-dir start
[root@bacsd01 ~]# /etc/init.d/bacula-sd start
[root@bacsd01 ~]# /etc/init.d/bacula-fd start
```

Bacula 的日志保存在 /var/log/bacula，如果发现启动有问题，可以查看相关日志。还可以在前台启动 Bacula 并打开调试模式：

```
[root@bacdir01 ~]# bacula-dir -f -d 99 -v
bacula-dir: dird.c:184-0 Debug level = 99
bacula-dir: mysql.c:167-0 mysql_init done
bacula-dir: mysql.c:188-0 mysql_real_connect done
bacula-dir: mysql.c:190-0 db_user=bacula db_name=bacula db_password=baculadb
```

5.4.6 Bacula 配置综述

Bacula 的各个组件，特别是控制器的配置文件比较复杂。当客户机和定义的备份作业比较多时，配置文件会变得非常冗长。此时可以将配置打散，将不同的资源分配到不同的

配置文件中。笔者在生产环境中使用了 Linux 下通用的 .d 目录 bacula-dir.d，目录里面包含如下文件和子目录：

```
[root@backupserver bacula]# ls -l bacula-dir.d/
总用量 68
-rw-r--r-- 1 root root 13448 11月 6 2014 clients.conf
-rw-r--r-- 1 420 420 228 7月 2 2013 fileset.conf
drwxr-xr-x 30 bacula bacula 4096 10月 21 2014 jobs
-rw-r--r-- 1 root root 329 6月 20 2013 restore.job.conf
-rw-r--r-- 1 root root 12941 11月 6 2014 storages.conf
```

此时需要在 bacula-dir.conf 里面将这些文件包含进来，加入如下配置：

```
@|sh -c 'for in in `find /etc/bacula/bacula-dir.d/ -type f -name \"*.conf\"`; do
echo @$in; done'
```

如果只是添加几个文件，@后面跟上文件名即可。这里使用 find 命令将 bacula-dir.d 目录下所有的 .conf 文件添加到主配置文件中。

Bacula 中各个组件之间的联系也需要通过配置文件里面的 Name 字段和 Password 字段来实现，因此要对其进行正确设置。Director、Console、FileDaemon 和 Storage 之间的联系如图 5-2 所示。

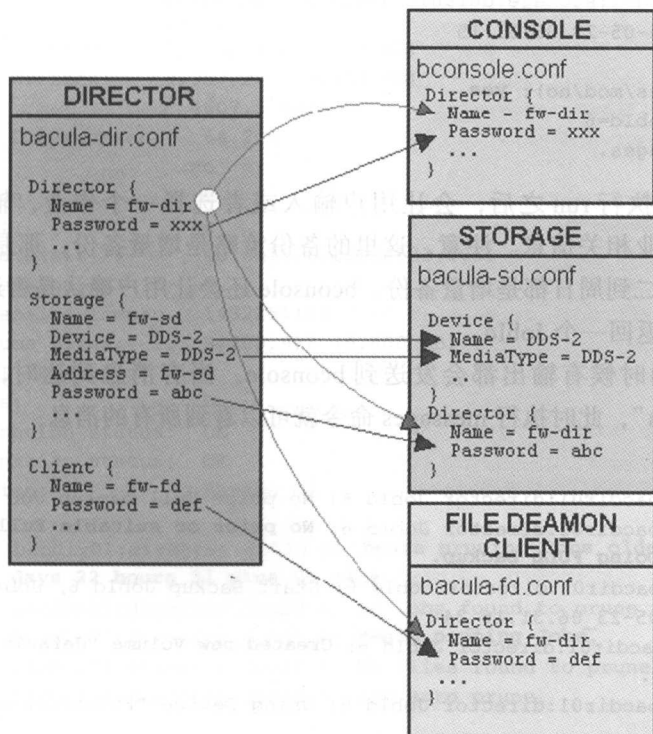


图 5-2 Bacula 各组件的验证关系

5.5 使用 Bacula 进行备份和恢复

5.5.1 执行备份

Bacula 所有的服务都启动之后，就可以进行一次手工备份了。启动 bconsole，执行 run 命令就可以启动备份作业，如下：

```
root@centos6 ~]# bconsole
Connecting to Director bacdir01:9101
1000 OK: bacdir01:director Version: 5.0.0 (26 January 2010)
Enter a period to cancel a command.
*run
A job name must be specified.
The defined Job resources are:
    1: bacsd01:default:etc
    2: bacdir01:restore:default
Select Job resource (1-2): 1
Run Backup job
JobName:  bacsd01:default:etc
Level:    Incremental
Client:   bacsd01
FileSet:  config:etc
Pool:     bacsd01:pool:default (From Job resource)
Storage:  bacsd01:storage:default (From Job resource)
When:     2015-05-23 06:31:15
Priority: 10
OK to run? (yes/mod/no): yes
Job queued. JobId=6
You have messages.
```

在 bconsole 中执行 run 之后，会让用户输入或者选择一个作业，输入作业的 ID 1，bconsole 会打印作业相关信息。注意，这里的备份策略是增量备份，那是因为之前定义备份计划的时候，周二到周日都是增量备份。bconsole 还会让用户确认是否运行，键入 yes 确认，此时控制器会返回一个 JobId。

作业在运行的时候有输出都会发送到 bconsole，当有消息到达时，bconsole 会提示“you have messages”，此时执行 messages 命令就可以看到所有的消息：

```
*messages
23-May 06:31 bacdir01:director JobId 6: No prior Full backup Job record found.
23-May 06:31 bacdir01:director JobId 6: No prior or suitable Full backup found in
    catalog. Doing FULL backup.
23-May 06:31 bacdir01:director JobId 6: Start Backup JobId 6, Job=bacsd01:default:
    etc.2015-05-23_06.31.16_24
23-May 06:31 bacdir01:director JobId 6: Created new Volume "default.6.20150523.0631"
    in catalog.
23-May 06:31 bacdir01:director JobId 6: Using Device "FileStorage"

23-May 06:31 bacsd01:storage JobId 6: Labeled new Volume "default.6.20150523.0631"
```

on device "FileStorage" (/opt/bacula/default).

23-May 06:31 bacsd01:storage JobId 6: Wrote label to prelabeled Volume "default.6.20150523.0631" on device "FileStorage" (/opt/bacula/default)

23-May 06:31 bacdir01:director JobId 6: **Max Volume jobs exceeded. Marking Volume "default.6.20150523.0631" as Used.**

23-May 06:31 bacsd01:storage JobId 6: Job write elapsed time = 00:00:02, Transfer rate = 4.678 M Bytes/second

23-May 06:31 bacdir01:director JobId 6: Bacula bacdir01:director 5.0.0 (26Jan10):

23-May-2015 06:31:21

Build OS: x86_64-redhat-linux-gnu redhat

JobId: 6

Job: bacsd01:default:etc.2015-05-23_06.31.16_24

Backup Level: **Full (upgraded from Incremental)**

Client: "bacsd01" 5.0.0 (26Jan10) x86_64-redhat-linux-gnu, redhat,

FileSet: "config:etc" 2015-05-21 20:27:11

Pool: "bacsd01:pool:default" (From Job resource)

Catalog: "bacdir01:mysql" (From Client resource)

Storage: "bacsd01:storage:default" (From Job resource)

Scheduled time: 23-May-2015 06:31:15

Start time: 23-May-2015 06:31:19

End time: 23-May-2015 06:31:21

Elapsed time: 2 secs

Priority: 10

FD Files Written: 1,270

SD Files Written: 1,270

FD Bytes Written: 9,214,998 (9.214 MB)

SD Bytes Written: 9,357,269 (9.357 MB)

Rate: 4607.5 KB/s

Software Compression: 64.2 %

VSS: no

Encryption: no

Accurate: no

Volume name(s): default.6.20150523.0631

Volume Session Id: 5

Volume Session Time: 1432254154

Last Volume Bytes: 9,400,938 (9.400 MB)

Non-fatal FD errors: 0

SD Errors: 0

FD termination status: OK

SD termination status: OK

Termination: Backup OK

23-May 06:31 bacdir01:director JobId 6: **Begin pruning Jobs older than 45 years 2 months 2 days 22 hours 31 mins 21 secs.**

23-May 06:31 bacdir01:director JobId 6: No Jobs found to prune.

23-May 06:31 bacdir01:director JobId 6: **Begin pruning Jobs.**

23-May 06:31 bacdir01:director JobId 6: No Files found to prune.

23-May 06:31 bacdir01:director JobId 6: End auto prune.

从消息输出可以看到, 此时进行的已经是一次全部备份, 因为这是用户第一次运行这

个作业，Bacula 从数据库 Catalog 中查找增量备份之前需要的全备份，发现信息不存在，所以自动将增量升级为全备。

笔者设置了自动打标，这里 Bacula 自动为 Volume 打上了 default.6.20150523.0631 的标签，如果没有设置，Bacula 会要求手动对 Volume 进行打标并加入到 Pool 之中。

同时，由于我们设置了一个 Volume 能够使用的最大备份作业为 1，因此 Bacula 会在将此 Volume 使用之后将其标志为“已使用”，这样其他的备份作业就不会再使用这个文件了。对于基于磁盘和文件的备份，一个 Volume 对应一个备份文件的设置使得备份目录条理清晰，是值得推荐的做法。

在备份工作完成之后，由于设置了 Auto Prune，Bacula 会尝试将过期的作业和文件清除。

查看 /opt/bacula/default，可以看到备份文件已经生成：

```
[root@bacsd01 ~]# ls /opt/bacula/default/default.6.20150523.0631
/opt/bacula/default/default.6.20150523.0631
```

此时再运行一次 run 命令，通过 status 命令查看是否进行了增量备份，如下：

```
*run
Automatically selected Catalog: baccd01:mysql
Using Catalog "baccd01:mysql"
A job name must be specified.
The defined Job resources are:
    1: bacsd01:default:etc
    2: baccd01:restore:default
Select Job resource (1-2): 1
Run Backup job
JobName:  bacsd01:default:etc
Level:    Incremental
Client:   bacsd01
FileSet:  config:etc
Pool:     bacsd01:pool:default (From Job resource)
Storage:  bacsd01:storage:default (From Job resource)
When:     2015-05-23 10:51:24
Priority: 10
OK to run? (yes/mod/no): yes
Job queued. JobId=7

*status
Status available for:
    1: Director
    2: Storage
    3: Client
    4: All
Select daemon type for status (1-4): 3
... ..
Terminated Jobs:
JobId  Level  Files  Bytes  Status  Finished  Name
```

```
=====
6 Full      1,270      9.214 M OK      23-May-15 06:31 bacsd01:default:etc
7 Incr      2          903   OK      23-May-15 10:54 bacsd01:default:etc
```

可以看到增量备份被正确地执行了。

5.5.2 文件恢复

恢复文件的命令是 `restore`，可启动 `bconsole` 来进行文件恢复，如下：

```
[root@bacdir01 ~]# bconsole
Connecting to Director bacdir01:9101
1000 OK: bacdir01:director Version: 5.0.0 (26 January 2010)
Enter a period to cancel a command.
*restore
Automatically selected Catalog: bacdir01:mysql
Using Catalog "bacdir01:mysql"
```

First you select one or more JobIds that contain files to be restored. You will be presented several methods of specifying the JobIds. Then you will be allowed to select which files from those JobIds are to be restored.

To select the JobIds, you have the following choices:

- 1: List last 20 Jobs run
- 2: List Jobs where a given File is saved
- 3: Enter list of comma separated JobIds to select
- 4: Enter SQL list command
- 5: Select the most recent backup for a client
- 6: Select backup for a client before a specified time
- 7: Enter a list of files to restore
- 8: Enter a list of files to restore before a specified time
- 9: Find the JobIds of the most recent backup for a client
- 10: Find the JobIds for a backup for a client before a specified time
- 11: Enter a list of directories to restore for found JobIds
- 12: Select full restore to a specified Job date
- 13: Cancel

Select item: (1-13):

恢复文件的流程是先选取一个 JobID，然后从对应的作业中选择恢复哪些文件。Restore 命令列出了一些选择 JobID 的方法，“Select the most recent backup for a client”应该是最方便的，它会选择一个客户单最近时间内进行的备份任务；也可以通过“List Jobs where a given File is saved”根据需要恢复的文件来查找 JobID。这里选择 2：

```
Select item: (1-13): 2
Automatically selected Client: bacsd01
Enter Filename (no path): issue
| JobId | Name          | StartTime          | JobType | JobStatus | JobFiles | JobBytes
```

```
| 6 | /etc/issue | 2015-05-23 06:31:19 | B | T | 1270 | 9214998
```

注意，输入文件的时候只需要输入文件名即可，不需要输入全路径。Bacula 顺利地找到了备份作业，其 JobID 为 6。然后选择 3，通过 JobID 来恢复文件：

```
Select item: (1-13): 3
Enter JobId(s), comma separated, to restore: 6
You have selected the following JobId: 6

Building directory tree for JobId(s) 6 ++++++
1,162 files inserted into the tree.
.....
cwd is: /
$
```

此时进入了文件选择模式，当前目录是 /，可以用 cd、ls、find 等命令来查找文件（输入 help 可以查看此模式下支持的所有命令）。使用 mark 和 unmark 来标记需要恢复的文件，当标记结束之后使用 done 命令，表示标记完成。

```
$ ls
etc/
$ cd etc
cwd is: /etc/
$ mark issue
1 file marked.
$ done
```

在这里，尝试标记并恢复 /etc/issue 这个文件。输入 done 命令之后，与备份作业类似，Bacula 会创建一个恢复作业，需要键入 yes 予以确认，如下：

```
JobName:          bacdir01:restore:default
Bootstrap:        /var/lib/bacula/bacdir01:director.restore.1.bsr
Where:            /tmp/restore
Replace:          always
FileSet:          config:etc
Backup Client:    bacsd01
Restore Client:   bacsd01
... ..
OK to run? (yes/mod/no): yes
Job queued. JobId=8
```

在输入 yes 确认的时候也可以输入 mod 命令，表示修改备份任务，一般可以用这个方法修改文件恢复目录，也就是 where 参数。同样，也可以使用 messages 查看作业进度。待作业完成之后，可以在其恢复目录 /tmp/restore 查看恢复的文件：

```
[root@centos6 ~]# cat /tmp/restore/etc/issue
CentOS release 6.6 (Final)
Kernel \r on an \m
```

可以看到 /etc/issue 这个文件已经被成功地恢复了。

在某个作业的文件由于过期而被自动清除的时候，Bacula 还是能够进行 restore 操作，只是此时不能够选择某个文件，只能恢复全部文件：

```
Enter JobId(s), comma separated, to restore: 8
You have selected the following JobId: 8
Building directory tree for JobId(s) 8 ...
```

```
For one or more of the JobIds selected, no files were found,
so file selection is not possible.
Most likely your retention policy pruned the files.
```

```
Do you want to restore all the files? (yes|no):
.....
```

5.6 Bacula 的使用和维护

5.6.1 Bconsole 的用法

除了运行 run 和 restore 进行备份和恢复之外，Bconsole 还有许多其他用法，可以用来监视和调试 Bacula 控制器。

比如已经演示过的 status 命令，它可以用来查看 Bacula 各个组件的状况。Status dir 命令能够打印当前运行的作业以及 24 小时内将要运行的作业，如下：

```
*status dir
Daemon started 23-May-15 16:55, 2 Jobs run since started.
Heap: heap=135,168 smbytes=61,872 max_bytes=190,325 bufs=176 max_bufs=206
```

Scheduled Jobs:

Level	Type	Pri	Scheduled	Name	Volume
Incremental	Backup	10	23-May-15 18:40	bacsd01:default:etc	*unknown*

Running Jobs:

```
Console connected at 23-May-15 18:08
No Jobs running.
=====
```

如果不加参数地运行 status，它会让用户选择需要查看的对象。

list 命令可以列出对象的基本情况，如 list jobs，用于查看所有已运行的备份作业；list files jobid=nn 则可以查看某个作业所备份的文件：

```
*list files jobid=7
+-----+
| Filename |
| /etc/bacula/ |
```



```
| /etc/bacula/bacula-dir.conf |
+-----+
|JobId|Name|StartTime|Type|Level|JobFiles|JobBytes|JobStatus|
|7|bacsd01:default:etc|2015-05-23 10:54:03|B|I|2|903|T|
```

此外, cancel 可以取消一个作业; delete 可以用来删除作业、Volume 和 Pool; help 命令可以看到所有支持的 console 命令。

5.6.2 使用 Bacula 进行文件验证

Bacula 会将文件的信息, 如文件属性和 md5 签名值等保存在数据库当中, 这使得 Bacula 能够用来验证系统文件。其基本的原理是首先对系统文件进行一次初始化的扫描, 将文件信息保存到 Catalog 里面, 然后再运行类型为 Verify 的作业进行对比:

```
FileSet {
    Name = "verify:etc"
    Include {
        Options {
            Verify=pins5
            Signature = MD5
        }
        File = /etc
    }
}
Job {
    Name = "Verify"
    Type = Verify
    Level = Catalog
    Client = bacsd01
    FileSet = "verify:etc"
    Messages = "bacdir01:messages:standard"
    Storage = "bacsd01:storage:default"
    Pool = "bacsd01:pool:default"
    Schedule = "Monthly:onMonday"
}
```

此时 FileSet 的定义需要加入 Verify 选项, 如 Signature 使用 MD5; Verify 设置为 pins5; 也可以 Signature 使用 SHA1, Verify 设置 pins1。将上述配置加入 bacula-dir.conf 中, 重启 Bacula 控制器, 然后使用 bconsole 运行此作业:

```
*run
.....
Run Verify job
JobName:      Verify
Level:        Catalog
... ..
OK to run? (yes/mod/no): mod
Parameters to modify:
```

```

1: Level
2: Storage
... ..
Select parameter to modify (1-9): 1
Levels:
1: Initialize Catalog
... ..
Select level (1-5): 1
Run Verify job
JobName:      Verify
Level:        InitCatalog
... ..
OK to run? (yes/mod/no): yes
Job queued. JobId=14

```

第一次运行此作业需要使用 mod 命令将作业的 level 调整为 InitCatalog, 在这个 level 下, 作业只是获得文件信息并保存到 Catalog, 不进行任何文件比较。

初始化数据库结束之后, 再在 /etc/issue 文件中添加如下一行数据:

```
[root@bacsd01 ~]# echo "test" >>/etc/issue
```

再次运行作业:

```

*run
.....
*messages
23-May 23:05 bacdir01:director JobId 20: File: /etc/issue
23-May 23:05 bacdir01:director JobId 20:      st_size differ. Cat: 75 File: 80
23-May 23:05 bacdir01:director JobId 20:      MD5 digest differs. File=
Pem+yL2oesucRXWEGGibAA Cat=4xJQdgLdLLX0g9buGQlMWg
23-May 23:05 bacdir01:director JobId 20: Bacula bacdir01:director 5.0.0
(26Jan10): 23-May-2015 23:05:08
      Build:                x86_64-redhat-linux-gnu redhat
      JobId:                 20
      Job:                   Verify.2015-05-23_23.05.03_06
      FileSet:               verify:etc
.....
      FD termination status: OK
      Termination:          Verify Differences

```

可以看到消息显示 /etc/issue 文件 MD5 值发生了变化, Verify 的结果为 “verify difference”。

5.6.3 Catalog 的维护和备份

如果没有合适地使用和维护, 随着作业的增加, Catalog 数据库保存的数据将越来越多, 比如对应一个备份工作, 其保存的所有文件信息都会记录到数据库里面, 其运行的效率和速度都会大幅下降。此时需要有一种机制, 持续地删除老的数据, 保证数据库不会过载。

Bacula 中的自动删除机制是通过设置 Retention 来实现的, 它使用了三种 Retention 设

置：File Retention、Job Retention 和 Volume Retention，这在之前都有介绍。

对于 MySQL 来说，删除数据库记录并不能有效地释放磁盘空间，一些空白的磁盘还会被其使用，我们可以将其数据 dump 一份，然后再导入到数据库，从而释放空白空间：

```
mysqldump -f --opt bacula > bacula.sql
mysql bacula < bacula.sql
rm -f bacula.sql
```

除此之外，还推荐对 Catalog 进行备份。Bacula 提供了一个备份数据库的脚本 `/usr/libexec/bacula/make_catalog_backup.pl`，这个脚本接受一个参数 Catalog 的名字，将 Catalog 数据库转存到 `/var/log/spool/bacula/catalog.sql`。在 Bacula 中，可以为 Catalog 创建自动备份作业，笔者的 Catalog 备份作业配置如下：

```
Job {
    Name = "bacdir01:Catalog"
    Type = Backup
    Schedule = "WeeklyCycleAfterBackup"
    Client = bacdir01
    RunBeforeJob = "/usr/libexec/bacula/make_catalog_backup.pl bacdir01:mysql"
    RunAfterJob = "/usr/libexec/bacula/delete_catalog_backup"
    RunAfterFailedJob = "/usr/libexec/bacula/delete_catalog_backup"
    FileSet = "Catalog"
    Storage = "bacsd01:storage:bacdir01:Catalog"
    Pool = "bacsd01:pool:catalog"
    Max Wait Time = 60
    Messages = "bacdir01:messages:standard"
}
```

这里使用的备份计划是 `WeeklyCycleAfterBackup`，这个计划设置备份的运行时间是在所有其他备份作业完成之后，避免数据库在更新的时候进行备份。`RunBeforeJob` 则设置成为 `make_catalog_back.pl`，在作业运行之前生成 SQL 文件，同时配置 `RunAfterJob`，将 SQL 文件删除。

Catalog 数据库中还有一个表有可能占用巨大空间，那就是 Log 表，它里面保存的是作业运行的日志。Bacula 在作业运行出错的时候，可能会产生巨量的错误日记，这些日志对 Catalog 本身的运行和维护都会造成一定影响，用户应该定期清理这个表，删除过期的日志。

5.7 备份的策略

使用 Bacula，用户可以轻松地备份一系列系统文件，但是对于一整套生产环境来说，备份工作不仅仅是使用好一套备份软件。用户不能恢复没有被备份的数据，但是怎么确认业务所需要的重要数据已经备份且备份成功了？备份的成功需要有一系列的措施和策略来保证。

5.7.1 备份什么

制定备份计划的第一步是找出生产环境中需要备份的数据。一般来说，这一步的解决方案就是回答问题：丢失哪些数据是我们所不能承受的？问题的答案就是需要备份的数据。与此同时，需要充分了解生产环境，了解重要的数据保存的位置，在哪一台服务器上，哪些文件之中，这样才能有效而准确地进行备份。

5.7.2 备份到哪里

按照备份数据保存位置的不同可以分为本地备份和异地备份。两种备份的区别同字面上的意思一样。好的备份策略应该考虑同时包括本地备份和异地备份。本地备份容易，恢复方便，同时比异地备份节省成本，是备份的首选方案；其缺点是由于备份和原始数据集在一起，容灾性较弱，当大的自然灾害发生的时候，备份和原始数据容易一起丢失。异地备份就是为了解决这一问题而出现的，它将数据备份到远离原始数据的异地机房，保证了地域上的分离性。异地备份的成本一般比较高，因此相对来说频率较本地备份低，是备份“备份数据”的最佳方案。

5.7.3 备份的时间

备份的时间和频率根据备份对象的不同而不同。不怎么改变的静态文件一般不需要太高的备份频率，而那些经常发生变化的文件则需要频繁地进行备份。

备份作业的开始时间也必须精心挑选。备份工作本身会消耗系统资源，影响系统性能，所以备份作业应该在系统负载比较小，机器比较空闲的时候进行，同时备份作业应该尽量不影响生产业务。一般推荐将备份作业设置在晚上运行，此时业务量比较小，机器的负载也较低。

5.7.4 测试和监控备份

只有数据能够被正确恢复，备份才有意义。为了保证备份数据的可用性，必须持续地对备份进行测试，以确保备份数据能够正常使用。测试的频率可以是一周一次，也可以是一个月一次，取决于备份作业的频率，一般来说备份越频繁，对其测试也应该越多。除此之外，对备份进行实时监控也是值得推荐的做法。Bacula 在备份失败的时候能够发送消息到相关人员，用户也可以通过监控系统，对失败的备份进行实时告警。

集群与存储

集群与存储几乎是所有业务稍有规模的公司都会用到的技术。本章会讲述高可用集群与负载均衡集群的搭建，以及存储的基本概念与配置。由于存储设备是一个高可用集群的基础，因此在本章，先从存储入手，然后进入高可用集群，最后讲解负载均衡。

6.1 存储的基本概念

对系统管理员来说，存储设备已经不再陌生，不少公司都购买了硬件存储设备，比如 NetApp、EMC 等，这些硬件存储设备一般分为 NAS 和 SAN 两种。NAS 的全称为 Network Attached Storage，是用于连接计算机的文件系统级别的存储，它支持多种协议，如 NFS、CIFS 等。SAN 的全称是 Storage Area Network，SAN 通过网络连接，将存储以块设备形式连接到计算机上，早期 SAN 需要使用 FC 光纤网络将存储挂接到服务器上，光纤设备价格高昂，使用成本较高，IP-SAN iSCSI 的出现，使得 SAN 可以通过以太网实现相同的功能，从而降低了 SAN 的使用成本。

现在的硬件存储，一些存储硬件厂商提供的产品可以同时拥有 NAS 和 SAN 功能，比如笔者之前使用的 NetApp FAS3140，但是一般来说，并不推荐使用这样的设备，因为多出来的功能可能一直到设备退役都用不上，而这些功能价格不菲。

6.2 SAN

这里不打算讲解 NAS 的搭建，因为 NFS、Samba 这些 Services 已经是系统管理员的基

本功。本节将讲解 SAN 的选择，以及 iSCSI 的配置。

6.2.1 SAN 的选择

当用户决定引入硬件存储设备的时候，首先需要决定是使用 FC-SAN 还是 IP-SAN，光纤通道通常的带宽为 4Gbit/s、8Gbit/s、16Gbit/s，可以提供很高的 I/O 吞吐量，而 iSCSI 是基于以太网的，现有商用产业中普遍的网络设备带宽为 1Gbit/s，而 10Gbit/s 价格较高，普及度较低，所以在性能表现上弱于光纤通道。但是 FC-SAN 需要光纤交换机的支持，且服务器上需要额外的 FC HBA 卡，相对来说，成本较高。而 IP-SAN 相对简单，只需要以太网即可。从现有的硬件能力来说，依然是 FC-SAN 的性能强于 IP-SAN，但是在未来万兆以太网普及的时候，IP-SAN 的性能会赶上 FC-SAN，不分伯仲。

所以在选择 FC-SAN 还是 IP-SAN 的时候，要根据现有的业务吞吐量，以及未来业务增长的趋势来评估选择。

6.2.2 iSCSI 的配置

iSCSI 的原理非常简单，就是将 SCSI 命令和 SCSI 数据包封装，加上 IP 报头通过 IP 协议层进行传输。

iSCSI 的配置分为两个部分：一个部分是 iSCSI Target，这就相当于 IP-SAN，是 iSCSI 的存储服务器，另一个部分是 iSCSI initiator，这是服务器用来挂接 IP-SAN 设备的软件。

iSCSI 有两种命名格式：一种是 iqn，一种是 EUI，后者使用较少，因为 EUI 命令不如 iqn 来得直观。iqn 的基本格式是 iqn.<YYYY-MM>.<reversed domain name>:<extra-name>，例如可以这样命名：iqn.2015-08.com.example:disk0。

1. 准备存储服务器

在这一步需要准备一个有大于 5GB 未分区的磁盘空间的虚拟机，然后在此之上创建一个 lvm group，过程如下：

```
[root@iscsi ~]# fdisk -l /dev/sda5

Disk /dev/sda5: 10.7 GB, 10740120064 bytes
255 heads, 63 sectors/track, 1305 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

[root@iscsi ~]# pvcreate /dev/sda5
Physical volume "/dev/sda5" successfully created
[root@iscsi ~]# vgcreate vg00 /dev/sda5
Volume group "vg00" successfully created
[root@iscsi ~]# vgs
VG      #PV #LV #SN Attr   VSize VFree
```

```

vg00 1 0 0 wz--n- 10.00g 10.00g
[root@iscsi ~]# lvcreate -L 5G -n example vg00
Logical volume "example" created
[root@iscsi ~]# lvs
LV VG Attr LSize Pool Origin Data% Meta% Move Log Cpy%Sync Convert
example vg00 -wi-a----- 5.00g
[root@iscsi ~]# ll /dev/vg00/example
lrwxrwxrwx 1 root root 7 Jun 24 12:00 /dev/vg00/example -> ../dm-0

```

2. 配置 iSCSI target

iSCSI target 配置并不复杂，在配置文件中也有一些注释的样例可以参考，这里配置一个基本的 iSCSI。

首先安装 `iscsi-target-utils` 包，然后在配置文件中配置 `iqn` 和打算作为 iSCSI 的卷，示例如下：

```

[root@iscsi ~]# yum install scsi-target-utils
[root@iscsi ~]# cat /etc/tgt/targets.conf | grep -v ^# | grep -v ^$
<target iqn.2015-06.com.example:iscsi-disk1>
    backing-store /dev/vg00/example
</target>

```

配置完成之后，重启 `tgtd` 服务，并将 `tgtd` 服务设为开机启动。

```

[root@iscsi ~]# /etc/init.d/tgtd start
Starting SCSI target daemon: [ OK ]
[root@iscsi ~]# chkconfig tgtd on

```

此时使用 `tgt-admin` 命令查看当前 iSCSI 卷的状态，可以看到 `online` 一项显示 `yes`，一切正常，可以进入下一步了，即为客户端挂接配置。

```

[root@iscsi ~]# tgt-admin --show
Target 1: iqn.2015-06.com.example:iscsi-disk1
System information:
    Driver: iscsi
    State: ready
I_T nexus information:
LUN information:
    LUN: 0
        Type: controller
        SCSI ID: IET 00010000
        SCSI SN: beaf10
        Size: 0 MB, Block size: 1
        Online: Yes
        Removable media: No
        Prevent removal: No
        Readonly: No
        Backing store type: null
        Backing store path: None
        Backing store flags:

```

```

LUN: 1
    Type: disk
    SCSI ID: IET      00010001
    SCSI SN: beaf11
    Size: 5369 MB, Block size: 512
    Online: Yes
    Removable media: No
    Prevent removal: No
    Readonly: No
    Backing store type: rdwr
    Backing store path: /dev/vg00/example
    Backing store flags:
Account information:
ACL information:
    ALL

```

3. 配置 iSCSI initiator

客户端配置也比较简单，大致分为三步：安装 `iscsi-initiator-utils` 包；通过 `iscsiadm` 命令发现 iSCSI 卷；挂接 iSCSI 卷。步骤如下：

```

[root@node1 ~]# yum install iscsi-initiator-utils -y
[root@node1 ~]# iscsiadm -m discovery -t sendtargets -p iscsi.example.com:3260
Starting iscsid: [ OK ]
192.168.0.7:3260,1 iqn.2015-06.com.example:iscsi-disk1

[root@node1 ~]# iscsiadm -m node -T iqn.2015-06.com.example:iscsi-disk1 -l
Logging in to [iface: default, target: iqn.2015-06.com.example:iscsi-disk1,
portal: 192.168.0.7,3260] (multiple)
Login to [iface: default, target: iqn.2015-06.com.example:iscsi-disk1, portal:
192.168.0.7,3260] successful.

```

从命令的输出可以看到 `iqn.2015-06.com.example:iscsi-disk1` 卷已经成功挂接，在系统中，用户可以使用 `fdisk` 命令，或者查看 `/dev/disk/`，或者使用 `iscsiadm` 命令查看 iSCSI 卷的挂接情况。

在系统中，这个 iSCSI 卷表现得就像一个本地磁盘，此时可以使用 `fdisk`、`parted` 等分区工具操作这个本地磁盘。这也就是常说的 SAN 是一种基于块设备的存储。

```

[root@node1 ~]# fdisk -l /dev/sdb

Disk /dev/sdb: 5368 MB, 5368709120 bytes
166 heads, 62 sectors/track, 1018 cylinders
Units = cylinders of 10292 * 512 = 5269504 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

[root@node1 ~]# ll /dev/disk/by-path/
total 0
lrwxrwxrwx 1 root root 9 Jun 25 22:47 ip-192.168.0.7:3260-iscsi-iqn.2015-06.com.

```



```

example:iscsi-disk1-lun-1 -> ../../sdb
lrwxrwxrwx 1 root root 9 Jun 25 22:44 pci-0000:00:10.0-scsi-0:0:0:0 -> ../../sda
lrwxrwxrwx 1 root root 10 Jun 25 22:44 pci-0000:00:10.0-scsi-0:0:0:0-part1 ->
../../sda1
lrwxrwxrwx 1 root root 10 Jun 25 22:44 pci-0000:00:10.0-scsi-0:0:0:0-part2 ->
../../sda2
lrwxrwxrwx 1 root root 10 Jun 25 22:44 pci-0000:00:10.0-scsi-0:0:0:0-part3 ->
../../sda3

[root@node1 ~]# iscsiadm -m session -P1
Target: iqn.2015-06.com.example:iscsi-disk1 (non-flash)
Current Portal: 192.168.0.7:3260,1
Persistent Portal: 192.168.0.7:3260,1
*****
Interface:
*****
Iface Name: default
Iface Transport: tcp
Iface Initiatorname: iqn.1994-05.com.redhat:b9558e2dc7fd
Iface IPaddress: 192.168.0.217
Iface HWaddress: <empty>
Iface Netdev: <empty>
SID: 1
iSCSI Connection State: LOGGED IN
iSCSI Session State: LOGGED_IN
Internal iscsid Session State: NO CHANGE

```

至此，一个 IP-SAN 的搭建完成了。你是否已经体验到一个 IP-SAN 的搭建与维护是如此的简单和轻松？

6.3 分布式文件系统与集群文件系统

6.3.1 分布式文件系统

尽管通过上一节了解到搭建一个基于软件的 IP-SAN 是如此简单，但是依然会有人想念 NFS，因为 NFS 的使用与搭建更为简单直观。但是 NFS 可扩展性低，且性能不高的缺陷使得它只能成为低端存储。试想一下，如果有 10 台服务器，要是都能像 LVM 一样将这 10 台服务器的存储在逻辑上组合成一个卷，在提升存储性能和空间的同时，又能以类似传统 NFS 网络挂载的方式使用，这该有多好啊！其实这就是分布式文件系统。

RedHat 官方提供了一套简单易用的分布式存储解决方案，即 GlusterFS。这是一种可扩展的分布式文件系统。下面来讲解如何配置使用 GlusterFS。

6.3.2 GlusterFS 的配置

1. 准备节点服务器

这里准备了 node1、node2、node3 三台虚拟机，每台机器划分出一个 5GB 的 lvm 逻辑

卷作为 glusterfs 的后端存储。

在 /etc/yum.repo.d/ 中建立一个以 .repo 为结尾的文件，内容如下，同时安装 glusterfs 软件包。

```
[root@node1 ~]# cat /etc/yum.repos.d/glusterfs.repo
# Place this file in your /etc/yum.repos.d/ directory

[glusterfs-epel]
name=GlusterFS is a clustered file-system capable of scaling to several petabytes.
baseurl=http://download.gluster.org/pub/gluster/glusterfs/3.4/LATEST/EPEL.repo/
epel-$releasever/$basearch/
enabled=1
skip_if_unavailable=1
gpgcheck=0

[glusterfs-noarch-epel]
name=GlusterFS is a clustered file-system capable of scaling to several petabytes.
baseurl=http://download.gluster.org/pub/gluster/glusterfs/3.4/LATEST/EPEL.repo/
epel-$releasever/noarch
enabled=1
skip_if_unavailable=1
gpgcheck=0

[glusterfs-source-epel]
name=GlusterFS is a clustered file-system capable of scaling to several petabytes.
- Source
baseurl=http://download.gluster.org/pub/gluster/glusterfs/3.4/LATEST/EPEL.repo/
epel-$releasever/SRPMS
enabled=0
skip_if_unavailable=1
gpgcheck=0

[root@node1 ~]# yum -y install xfsprogs.x86_64 glusterfs-server.x86_64
```

安装 glusterfs 时，使用了一个 LVM 卷作为存储方式，同时将其格式化为 xfs 格式，并将其挂在到某个目录下。示例如下：

```
[root@node1 ~]# mkfs.xfs /dev/mapper/vg00-lv00
meta-data=/dev/mapper/vg00-lv00  isize=256    agcount=4, agsize=327936 blks
        =                               sectsz=512   attr=2, projid32bit=0
data      =                               bsize=4096   blocks=1311744, imaxpct=25
        =                               sunit=0     swidth=0 blks
naming    =version 2                     bsize=4096   ascii-ci=0
log       =internal log                  bsize=4096   blocks=2560, version=2
        =                               sectsz=512   sunit=0 blks, lazy-count=1
realtime  =none                           extsz=4096   blocks=0, rtextents=0
[root@node1 ~]# mkdir /node1-data
[root@node1 ~]# mount /dev/mapper/vg00-lv00 /node1-data/
```

以上步骤在 node2 和 node3 上重复。

2. 配置节点服务器

在 node1、node2、node3 上启动 glusterd 进程，命令如下：

```
[root@node1 ~]# /etc/init.d/glusterd start
[root@node2 ~]# /etc/init.d/glusterd start
[root@node3 ~]# /etc/init.d/glusterd start
```

在 glusterfs 中没有中心节点的概念，所以可以在任意的一个节点配置 GlusterFS 信息，这里选择 node1。

```
[root@node1 ~]# gluster peer probe node2.example.com
peer probe: success
```

```
[root@node1 ~]# gluster peer probe node3.example.com
peer probe: success
```

当 node2/node3 被加入 cluster 之后，可以看到集群中 node 的状态，如下：

```
[root@node1 ~]# gluster peer status
Number of Peers: 2

Hostname: node3.example.com
Uuid: 19ce3196-706d-4c25-93c0-f012758a6125
State: Peer in Cluster (Connected)

Hostname: node2.example.com
Uuid: 4f5dc30c-ebb5-4419-b08b-034c7c393db1
State: Peer in Cluster (Connected)
```

此时就可以创建一个分布式的卷了，GlusterFS 默认提供分布式卷。创建卷的步骤是先创建出卷，然后启用之，最后在其他机器上挂载此卷。

待卷创建完成之后，使用 volume info 命令可以看到这个卷的类型是分布式，节点之间的传输方式是 TCP。

```
[root@node1 ~]# gluster volume create example-vol node1.example.com:/node1-data/
export node2.example.com:/node2-data/export node3.example.com:/node3-data/
export
```

```
[root@node1 ~]# gluster volume info example-vol
Volume Name: example-vol
Type: Distribute
Volume ID: b5b2042f-a0e7-4aee-8376-51b40a313236
Status: Created
Number of Bricks: 3
Transport-type: tcp
Bricks:
Brick1: node1.example.com:/node1-data/export
Brick2: node2.example.com:/node2-data/export
Brick3: node3.example.com:/node3-data/export
```

```
[root@node1 ~]# gluster volume start example-vol
volume start: example-vol: success
```

3. 使用和测试

下面在一台空闲 vm 上装上 glusterfs-fuse 来测试使用这个卷。因为 GlusterFS 是一个无中心化的分布式存储，所以可以挂接 node1/node2/node3 中的任意节点，这里挂接的是 node2。

```
[root@localhost ~]# yum -y install glusterfs-fuse.x86_64
[root@localhost ~]# mount.glusterfs node2:/example-vol /mnt/
[root@localhost ~]# df -h
```

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/sda2	20G	1.4G	18G	8%	/
tmpfs	931M	0	931M	0%	/dev/shm
/dev/sda1	93M	32M	57M	36%	/boot
node2:/example-vol	15G	97M	15G	1%	/mnt

在这个目录里创建了 10 000 个文件，然后去 node1、node2、node3 上查看，就会发现在三个节点中，存储了近似相等的文件，至此 GlusterFS 的卷配置成功了。

```
[root@localhost ~]# touch /mnt/file{1..10000}
[root@node1 ~]# ls -l /node1-data/export/ | wc -l
3441
[root@node2 ~]# ls -l /node2-data/export/ | wc -l
3238
[root@node3 ~]# ls -l /node3-data/export/ | wc -l
3324
```

6.4 高可用集群

集群分为两种：高可用集群和负载均衡集群，尽管负载均衡集群可以达到高可用的目的，但是两者面对的业务类型不一样，负载均衡集群多数情况下用于 Web 前端，高可用集群多数用于数据库这种后端类型的服务。这里以 RedHat 提供的 HA Cluster 为例，讲解高可用集群的组成与配置。

6.4.1 Red Hat HA Cluster 简介

图 6-1 是 Red Hat HA 物理架构，简单来说，分为 Cluster Nodes、Shared storage、Network power switch、Ethernet switch 等几个部分。除了 Red Hat，大部分的 HA 软件架构（比如 Veritas 的 VCS）也是类似的结构。在高可用集群中，节点通常以一主一备或一主多备的方式出现，有时候也会出现多机互备的情况，提高资源利用率。

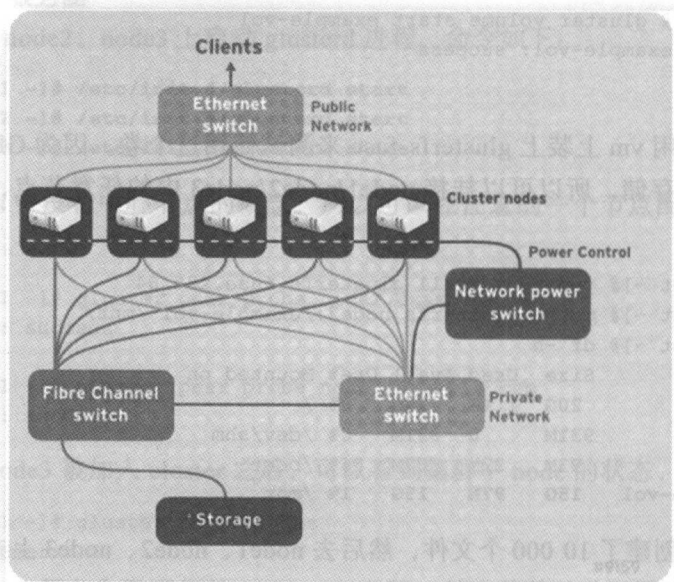


图 6-1 Red Hat HA 的物理架构

Red Hat HA cluster 中包含如下的一些软件。

- ❑ Corosync: 用于节点间的心跳检测。
- ❑ Cman: 集群管理器。
- ❑ Fenced: 栅设备, 用来判断不可用节点, 同时将其剔除 cluster, 防止出现脑裂的情况。
- ❑ Luci: 用来配置集群的 Web 页面。
- ❑ Modclusterdr Ricci: 与 luci 之间通信的代理程序。
- ❑ Rgmanager: 控制集群资源。
- ❑ Ricci: 接受 Luci 配置文件。
- ❑ DLM rgmanager: 通信进程。
- ❑ clvmd: 集群逻辑卷管理。

但是注意, Red Hat HA 最大支持 16 节点, 同时要尽量避免双机 HA, 因为使用了投票仲裁机制, 所以如果需在生产环境中使用, 建议使用三台机器以上做 HA Cluster。

6.4.2 配置一个高可用的 Apache 集群

这里依然用三个节点作为集群基础环境, 下面以配置一个高可用的 Apache 集群为例, 讲解高可用集群的配置与使用。

1. 安装 HA 软件包

首先，依然是安装相关软件包，确保所有节点机器上的 NetworkManager 服务被关闭且没有被设置为自动启动，并确保 iptables 规则清空。然后在 node1、node2、node3 上安装 High Availability 软件包组和 Apache 软件包，同时开启 cman 和 rgmanager 服务。示例如下：

```
[root@node1 ~]# /etc/init.d/NetworkManager stop;chkconfig NetworkManager off
[root@node1 ~]# yum -y groupinstall 'High Availability'
[root@node1 ~]# chkconfig cman on;/etc/init.d/cman start
[root@node1 ~]# chkconfig rgmanager on;/etc/init.d/rgmanager start
[root@node1 ~]# yum -y install httpd
```

在 node2、node3 上重复上述步骤。

这里选择使用 Web 界面配置整个 HA 集群，所以需要安装管理工具 Luci。Luci 可以安装在任意一个 node 上或其他网络中能够和这个集群通信的机器上。这里选择在 node1 上安装 Luci，命令如下：

```
[root@node1 ~]#yum -y groupinstall 'High Availability Management'
```

2. 配置 Web 管理工具 Luci

Luci 与节点上的 ricci 服务进行通信，以实现配置文件的下发和同步，所以这里首先在 node1、node2、node3 上将 ricci 服务启动起来，然后再进行 Luci 服务的配置。

在 node1、node2、node3 上启动 ricci 服务，基本过程就是给 ricci 用户设置密码，然后启动服务。node1 上的示例如下：

```
[root@node1 ~]# echo redhat | passwd --stdin ricci
[root@node1 ~]# chkconfig ricci on
[root@node1 ~]# /etc/init.d/ricci start
```

在 node2、node3 上重复上述步骤。

Luci 服务的配置很简单，只需要将服务启动，然后在浏览器中访问 <http://node1.example.com:8084> 就可以进行配置了。示例如下：

```
[root@node1 ~]# /etc/init.d/luci start
[root@node1 ~]# chkconfig luci on
```

Luci 采用的是系统用户的认证方式，所以这里可以使用 root 用户登录。在生产环境中，建议单独建立一个用户专门负责 Luci 服务的登录认证，以确保系统安全。Luci 的登录界面如图 6-2 所示。

3. 配置共享存储

共享存储是整个集群的基石，这里选择 iSCSI 作为后端共享存储，如何构建一个 iSCSI 存储请参考 6.2.2 节，这里只讲述存储的挂接步骤。

在 node1、node2、node3 上先挂接测试 iSCSI 存储，步骤如下，请在 node2、node3 上重复下列步骤。

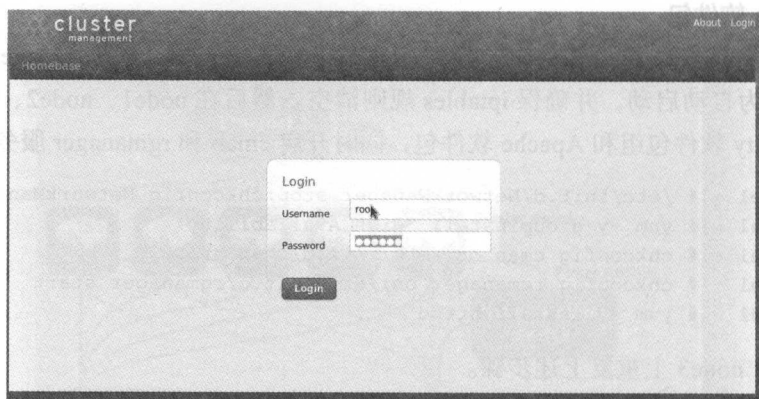


图 6-2 Luci 登录界面

```
[root@node1 ~]# yum install iscsi-initiator-utils -y
[root@node1 ~]# iscsiadm -m discovery -t sendtargets -p iscsi.example.com:3260
Starting iscsid: [ OK ]
192.168.0.7:3260,1 iqn.2015-06.com.example:iscsi-disk1

[root@node1 ~]# iscsiadm -m node -T iqn.2015-06.com.example:iscsi-disk1 -l
Logging in to [iface: default, target: iqn.2015-06.com.example:iscsi-disk1,
portal: 192.168.0.7,3260] (multiple)
Login to [iface: default, target: iqn.2015-06.com.example:iscsi-disk1, portal:
192.168.0.7,3260] successful.
```

待在三个节点上存储挂载都没有问题以后，在 node1 上对这块挂载的 iSCSI 盘进行分区操作，并转换成 LVM 分区，最后将分区挂到 /var/www/html 上，这是 Apache 的默认页面存储目录，具体分区和转换成 LVM 的操作这里不再重复。

```
[root@node1 ~]# fdisk /dev/mapper/clusterstorage
[root@node1 ~]# partprobe
[root@node1 ~]# mkfs.ext4 /dev/mapper/clusterstorage1
[root@node1 ~]# mount -t ext4 /dev/mapper/clusterstorage1 /var/www/html/
```

分区完成之后，在 node2、node3 上运行命令 partprobe 就可以看到新分区 /dev/mapper/clusterstorage1 了。

4. 测试资源

资源是高可用集群的基本组成部分，其中包括 httpd 服务、IP、共享存储等。在使用这些资源之前，需要在所有节点上先进行测试，确保它们在所有节点上都可用，这样才能使得资源在节点间切换正常。

首先测试在节点上加上一个 IP 是否能正常工作。这个 IP 也就是浮动 IP，它会跟随集群切换而出现在活动的节点上。比如：使用 ip add 命令在 eth1 上加上一个 IP 172.16.10.50，然后从其他节点上 ping 这个 IP，看是否能通，能通则说明 IP 添加没有问题，在 node1、

node2、node3 上全部测试通过之后进入下一步资源测试。注意，待测试完成之后要将加入的 IP 删掉，否则会造成后面的资源配置冲突。

```
[root@node1 ~]# ip addr add dev eth1 172.16.10.50/24
[root@node1 ~]# ip add show eth1
[root@node1 ~]# ip addr show eth1 | grep 172
    inet 172.16.26.1/24 brd 172.16.26.255 scope global eth1
inet 172.16.26.50/24 scope global secondary eth1
[root@node1 ~]# ip addr del 172.16.10.50/24 dev eth1
```

其次测试 httpd 服务是否正常。在 node1 上启动 httpd，可能会发现如下错误，这是 SELinux 的问题，可以选择关闭 SELinux，或者配置 SELinux 的上下文。

```
[root@node1 ~]# service httpd start
Starting httpd: Syntax error on line 292 of /etc/httpd/conf/httpd.conf:
DocumentRoot must be a directory
[FAILED]
[root@node1 ~]# chcon -R -t httpd_sys_content_t /var/www/html/
[root@node1 ~]# service httpd restart
Stopping httpd:
Starting httpd:
[ OK ]
```

然后在 /var/www/html 里放入一个非常简单的页面“Hello World”，同样，在测试完成之后要将 httpd 服务停止，卸载磁盘以防止后面产生配置冲突。

```
[root@node1 ~]# echo "Hello World" > /var/www/html/index.html
[root@node1 ~]# ls -Z /var/www/html/index.html
-rw-r--r--. root root unconfined_u:object_r:httpd_sys_content_t:s0 /var/www/html/
index.html
[root@node1 ~]# elinks -dump http://172.16.10.50
Hello World
[root@node1 ~]# /etc/init.d/httpd stop
Stopping httpd:
[ OK ]
[root@node1 ~]# umount /var/www/html/
```

资源测试完成以后，可以进入下一步配置了。

5. 配置集群组

打开 Luci，首先建立一个 cluster，这里是点击 create，不是点击 add，add 是添加一个已经存在的 cluster（如图 6-3 所示）。

然后在 cluster name 中填写名字，这里没有特别需求，最后填入 node1、node2、node3 的信息即可（如图 6-4 所示）。

配置好 cluster 之后开始配置资源，顺序是先添加浮动 IP，然后添加文件系统，最后添加服务。

首先，点击 Add Resource，选择 IP Address，具体配置如图 6-5 所示。

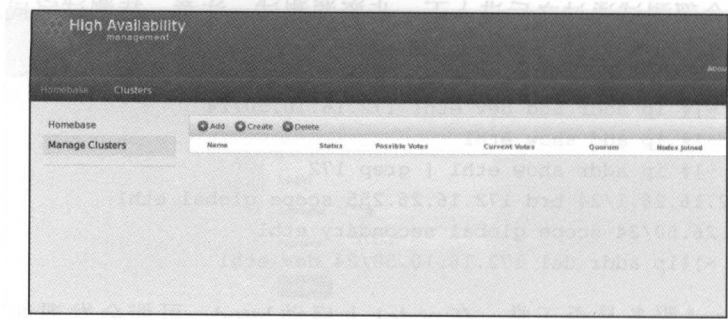


图 6-3 集群控制面板

Create New Cluster

Cluster Name:

☒ Use the Same Password for All Nodes

Node Name	Password	Ricci Hostname	Ricci Port
rate.cluster26.example.com	*****	node1.private.cluster26.example.com	11111
node2.private.cluster26.example.com	*****	node2.private.cluster26.example.com	11111
node3.private.cluster26.example.com	*****	node3.private.cluster26.example.com	11111

☒ Download Packages
☐ Use Locally Installed Packages

☒ Reboot Nodes Before Joining Cluster

☒ Enable Shared Storage Support

图 6-4 集群节点基本配置选项

Add Resource to Cluster

IP Address:

IP Address

IP Address:

Netmask Bits (optional):

Monitor Link: ☒

Disable Updates to Static Routes: ☒

Number of Seconds to Sleep After Removing an IP Address:

图 6-5 集群浮动 IP 配置

然后重复 Add resource，选择 Filesystem，具体配置如图 6-6 所示。

The screenshot shows the 'Add Resource to Cluster' dialog box with the 'Filesystem' resource selected. The configuration fields are as follows:

Field	Value
Name	webfs
Filesystem Type	ext4
Mount Point	/var/www/html/
Device, FS Label, or UUID	/dev/mapper/clusterstorage1
Mount Options	
Filesystem ID (optional)	
Force fsck	<input type="checkbox"/>
Use Quick Status Checks	<input checked="" type="checkbox"/>
Reboot Host Node if Unmount Fails	<input checked="" type="checkbox"/>

Buttons: Submit, Cancel

图 6-6 集群共享磁盘配置

加入 httpd 资源，配置如图 6-7 所示。

The screenshot shows the 'Add Resource to Cluster' dialog box with the 'Apache' resource selected. The configuration fields are as follows:

Field	Value
Name	apache
Server Root	/etc/httpd
Config File	conf/httpd.conf
httpd Options	
Shutdown Wait (seconds)	3

Buttons: Submit, Cancel

图 6-7 集群应用配置

所有的 resource 配置完成之后的界面如图 6-8 所示。

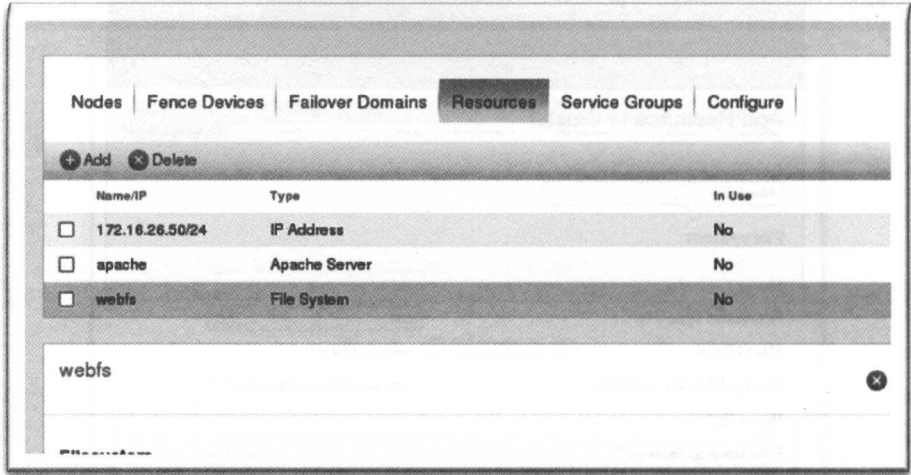


图 6-8 集群资源控制面板

资源配置完成之后，再对 Failover Domains 进行配置，配置非常简单，如图 6-9 所示。注意这里的 priority 决定了资源组在哪个机器上率先启动。

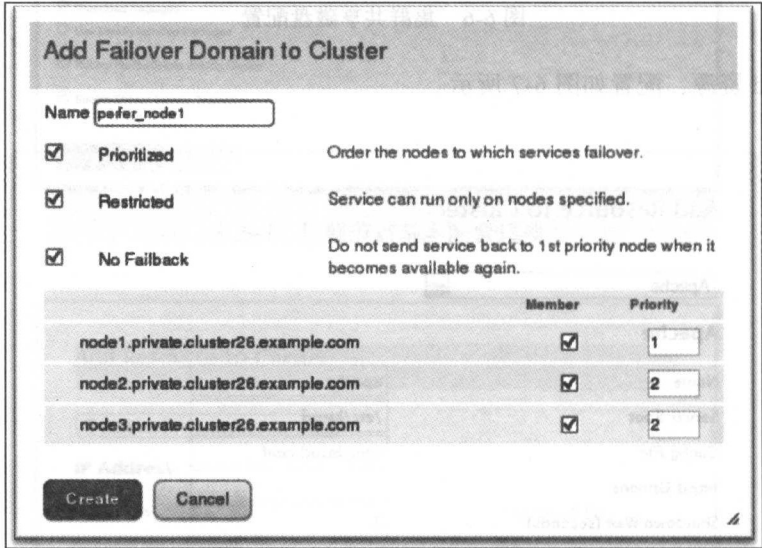


图 6-9 集群 Failover 规则配置

最后配置 Service Group，将前面配置的 resource 逐一加入 Service Group，注意这里的资源顺序，需要先启动 filesystem 资源，然后再启动 httpd 资源（如图 6-10 至图 6-12 所示）。

Add Service Group to Cluster

Service Name: web

Automatically Start This Service: ☒

Run Exclusive: ☒

Failover Domain: prefer_web

Recovery Policy: Relocate

Restart Options

Maximum Number of Restart Failures Before Relocating:

Length of Time in Seconds After Which to Forget a Restart:

Add Resource

Submit Cancel

图 6-10 集群服务组配置

Add Service Group to Cluster

Service Name: web

Automatically Start This Service: ☒

Run Exclusive: ☒

Failover Domain: prefer_web

Recovery Policy: Relocate

Restart Options

Maximum Number of Restart Failures Before Relocating:

Length of Time in Seconds After Which to Forget a Restart:

Add Resource to Service

-- Select a Resource Type --

-- Global Resources --

172.16.10.50/24

apache

webfs

-- Select a Resource Type --

Apache

DRBD Resource

Filesystem

GFS2

IP Address

HA LVM

MySQL

NFS/CIFS Mount

NFS

Add Resource

Submit Cancel

图 6-11 集群服务组资源类型

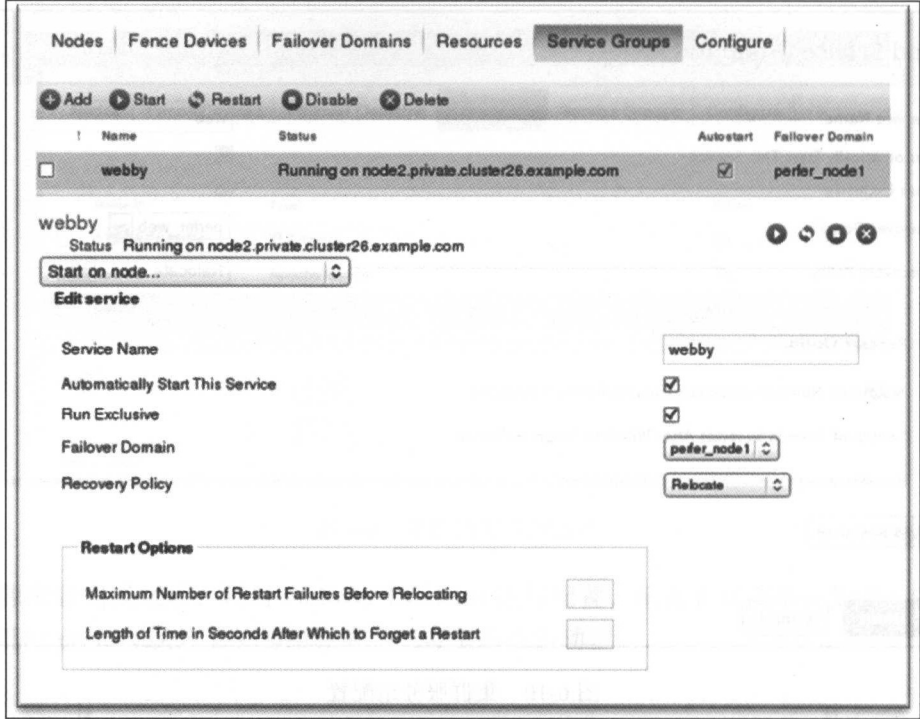


图 6-12 集群服务器配置面板

至此整个集群组配置完成，可以进行集群切换测试了。

6. 切换集群组测试

先使用 `clustat` 命令查看整个集群状态，代码如下，可以看到此时一切正常。

```
[root@node1 ~]# clustat
Cluster Status for cluster1 @ Fri Oct 23 15:02:02 2015
Member Status: Quorate

Member Name                                     ID   Status
-----
node1.private.cluster26.example.com             1 Online, Local, rgmanager
node2.private.cluster26.example.com             2 Online, rgmanager
node3.private.cluster26.example.com             3 Online, rgmanager

Service Name                                     Owner (Last) State
-----
service:webby
node2.private.cluster26.example.com             started
```

切换 `webfs` 资源组到 `node3` 上，大约在 10 秒后，再用 `clustat` 命令查看集群状态。此时 `webby` 切换到了 `node3` 上，同时我们始终可以访问 `http://172.16.10.50`。

```
[root@node1 ~]# clusvcadm -r web -m node3.private.cluster10.example.com
[root@node1 ~]# clustat
Cluster Status for cluster1 @ Fri Oct 23 15:20:22 2015
Member Status: Quorate
```

Member Name	ID	Status
node1.private.cluster26.example.com	1	Online, Local, rgmanager
node2.private.cluster26.example.com	2	Online, rgmanager
node3.private.cluster26.example.com	3	Online, rgmanager

Service Name	Owner (Last) State
service:webby	
node3.private.cluster26.example.com	started

```
[root@node1 ~]# elinks -dump http://172.16.10.50
Hello World
```

这样就完成了一个 Apache 的高可用集群组。一般来说，这种高可用集群组多数出现在数据库服务上，而对于 Apache 这样的 Web 应用，多采用负载均衡集群，下一节将讲解如何配置负载均衡集群。

6.5 负载均衡集群

6.5.1 HAProxy 负载均衡

HAProxy 是一款开源的负载均衡软件，可用于 4 层和 7 层转发，从主流应用来看，使用 HAProxy 作为 7 层 HTTP 转发的场景非常多，因为它支持 session、header rewrite、双机热备、虚拟主机等功能。在性能方面，官方宣称其可支持 10GB 并发量，所以 HAProxy 非常适合用于大并发、大流量的使用场景。本节将使用一台 HAProxy 作为前端，两台 Apache 作为后端来演示如何安装配置，并假设两台 Apache 服务器的 IP 地址分别为 192.168.10.18/19。

两台 Apache 服务器使用 yum 安装后，分别在 /var/www/html 目录下创建 index.html，文件内容分别为“Server1”和“Server2”，用于后面使用 HAProxy 进行负载均衡后区分实际访问到的不同主机，该步骤完成后，启动 httpd 服务。

使用 yum 安装 HAProxy 非常简便，但是目前 CentOS 默认源中提供的版本为 1.5，如果想要安装更新的版本可使用源码包编译安装。示例如下：

```
[root@192 ~]# yum install haproxy
Loaded plugins: fastestmirror
Setting up Install Process
Loading mirror speeds from cached hostfile
* base: ftp.sjtu.edu.cn
```

```

* extras: ftp.sjtu.edu.cn
* updates: ftp.sjtu.edu.cn
Resolving Dependencies
--> Running transaction check
---> Package haproxy.x86_64 0:1.5.4-2.el6_7.1 will be installed
--> Finished Dependency Resolution

```

Dependencies Resolved

```

=====
Package Arch Version Repository Size
=====
Installing:
haproxy x86_64 1.5.4-2.el6_7.1 updates 792 k

```

Transaction Summary

```

=====
Install 1 Package(s)

Total download size: 792 k
Installed size: 2.4 M
Is this ok [y/N]: y
Downloading Packages:
haproxy-1.5.4-2.el6_7.1.x86_64.rpm | 792 kB 00:00
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
Installing : haproxy-1.5.4-2.el6_7.1.x86_64 1/1
Verifying : haproxy-1.5.4-2.el6_7.1.x86_64 1/1

```

```

Installed:
haproxy.x86_64 0:1.5.4-2.el6_7.1

```

Complete!

安装完成后，修改配置文件 `/etc/haproxy/haproxy.cfg`，添加下面配置中加粗的部分。该部分配置的作用是：

- ❑ 让 HAProxy 监听 8080 端口，当使用 `http://IP:8080/stats` 访问时，输入用户名 `admin`、密码 `admin`，从而进入 HAProxy 状态统计页面。
- ❑ 让 HAProxy 监听 80 端口，当使用 `http://IP:8080` 访问时，负载均衡地将请求发送到后端 `192.168.10.18/19` 服务器上。通过刷新浏览器可以看出，随着每次的刷新，页面上依次显示为“Server1”和“Server2”，这说明 HAProxy 工作正常。

修改完成后，使用 `/etc/init.d/haproxy start` 启动 HAProxy，并访问 `stats` 页面，该状态页

可以显示 HAProxy 的当前性能数据，如下：

```
defaults
    mode                http
    log                 global
    option              httplog
    option              dontlognull
    option http-server-close
    option forwardfor   except 127.0.0.0/8
    option              redispatch
    retries             3
    timeout http-request 10s
    timeout queue       1m
    timeout connect     10s
    timeout client      1m
    timeout server      1m
    timeout http-keep-alive 10s
    timeout check       10s
    maxconn             3000

listen status *:8080
    mode http
    stats uri /stats
    stats auth admin:admin
```

```
#-----
# main frontend which proxys to the backends
#-----
frontend main *:80
    default_backend      app

#-----
# round robin balancing between the various backends
#-----
backend app
    balance      roundrobin
    server app1 192.168.10.18:80 check
    server app2 192.168.10.19:80 check
```

HAProxy 的运行状态如图 6-13 所示。

通过以上的配置，我们成功地使用 HAProxy 作为负载均衡器将需求分流给了后端的服务器。但是这里的演示中，HAProxy 是一个单点，这对于很多应用来说是不可接受的。不过，读者可以通过引入 keepalived 来解决这个问题。

6.5.2 Nginx 负载均衡

与 HAProxy 有所不同，Nginx 本身是一个 HTTP 服务器，同时也提供了负载均衡的功能。本节将演示 Nginx 的安装方式并使用 Nginx 作为负载均衡设备代替上一节中的 HAProxy。Nginx 的安装过程稍微复杂一点，因为 CentOS 官方并没有提供可供 yum 安装的

包，而编译安装过程中的依赖关系又比较复杂，每个人在实际编译安装过程中遇到的依赖关系可能不完全一样。

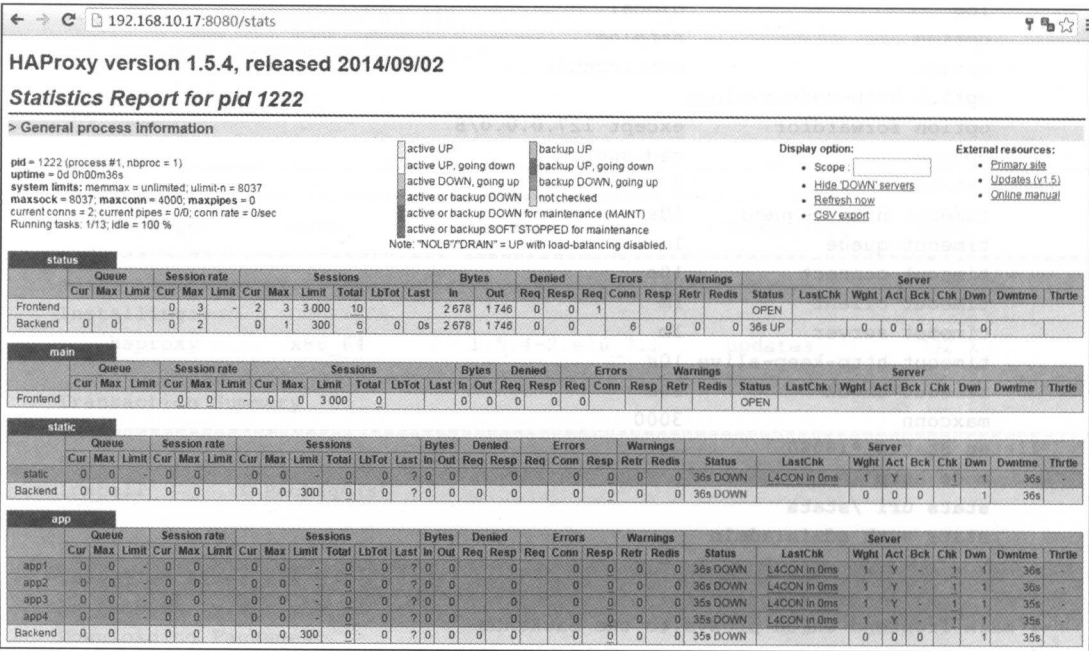


图 6-13 HAProxy 的运行状态

读者可通过 <http://nginx.org/en/download.html> 下载到最新版本的 Nginx，截至笔者撰写本章时，最新版本为 1.8.1。示例如下：

```
yum remove haproxy #如果使用Nginx替代HAProxy，则删除HAProxy包

#下载Nginx源码包并解压进入目录
wget http://nginx.org/download/nginx-1.8.1.tar.gz
tar zxvf nginx-1.8.1.tar.gz
cd nginx-1.8.1

#安装必要的依赖包
yum install gcc pcre-devel zlib-devel openssl-devel

#编译并安装到/usr/local/nginx
./configure --prefix=/usr/local/nginx && make && make install

修改配置文件 conf/nginx.conf，并启动 Nginx 服务：
cat conf/nginx.conf

worker_processes 1;
```

```

events {
    worker_connections 1024;
}

http {
    include mime.types;
    default_type application/octet-stream;

    sendfile on;

    keepalive_timeout 65;

    upstream app_pool {
        server 192.168.10.18:80;
        server 192.168.10.19:80;
    }

    server {
        listen 80;
        server_name localhost;
        location / {
            root html;
            index index.html index.htm;
            proxy_pass http://app_pool;
        }

        error_page 500 502 503 504 /50x.html;
        location = /50x.html {
            root html;
        }
    }
}

#启动Nginx
./sbin/nginx

```

使用 `http://IP` 访问 Nginx，并不断刷新浏览器，若页面上依次显示为“Server1”和“Server2”，则说明 Nginx 工作正常。

6.5.3 LVS 负载均衡

LVS 负载均衡是国人的贡献，它工作在网络 4 层，最大的优点是转发效率极高，大多数情况下性能仅受限于硬件上的网卡性能。

LVS 目前有 4 种工作模式：NAT、DR、TUNNEL 和 Full-NAT，其中最常用的是 NAT 和 DR 模式，TUNNEL 模式使用场景极少，而 Full-NAT 模式用于跨网段（或是跨机房），所以使用场景并不典型。本节将从实用出发，演示 NAT 模式和 DR 模式。

NAT 模式和之前的 HAProxy、Nginx 类似，该模式就是“代理模式”，即：所有的访问流量和返回流量都通过负载均衡设备，LVS 的 NAT 工作模式是所有模式中最简单、最易配置的。这里将使用三台服务器做演示，一台为 LVS 负载均衡器，另外两台运行 httpd 服务。

在一台服务器上安装 ipvsadm，作为 LVS 负载均衡器，该服务器有两块网卡，IP 分别为 192.168.109.131 和 192.168.1.5，其中 192.168.1.5 用于对外提供负载均衡服务。

```
[root@192 ~]# yum install ipvsadm
Loaded plugins: fastestmirror
Setting up Install Process
Loading mirror speeds from cached hostfile
  * base: mirrors.aliyun.com
  * extras: mirrors.aliyun.com
  * updates: mirrors.aliyun.com
Resolving Dependencies
--> Running transaction check
---> Package ipvsadm.x86_64 0:1.26-4.el6 will be installed
--> Finished Dependency Resolution
```

Dependencies Resolved

```
=====
Package                Arch      Version      Repository      Size
=====
Installing:
  ipvsadm              x86_64    1.26-4.el6    base             42 k
```

Transaction Summary

```
=====
Install                1 Package(s)
```

Total download size: 42 k

Installed size: 78 k

Is this ok [y/N]: y

Downloading Packages:

ipvsadm-1.26-4.el6.x86_64.rpm | 42 kB 00:00

Running rpm_check_debug

Running Transaction Test

Transaction Test Succeeded

Running Transaction

Installing : ipvsadm-1.26-4.el6.x86_64 1/1

Verifying : ipvsadm-1.26-4.el6.x86_64 1/1

Installed:

ipvsadm.x86_64 0:1.26-4.el6

Complete!

在另外两台服务器上安装 httpd 服务，并启动（可以沿用前两节的 httpd 环境）。假设这两台服务器的 IP 地址分别为 192.168.109.129 和 192.168.109.130，现在要将这两台服务器的默认网关设置为 LVS 的 IP：192.168.109.131，如下：

```
route delete default #删除原先的默认路由
route add default gw 192.168.109.131 #添加新的默认路由
```

在 LVS 节点上继续做以下配置，可以将其写为脚本，方便每次使用，执行该脚本后，就会在 LVS 服务器上创建一个以 192.168.1.5:80 为入口的负载均衡服务。

```
[root@192 ~]# cat lvs-nat.sh
#!/bin/bash
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects
echo 0 > /proc/sys/net/ipv4/conf/default/send_redirects
echo 0 > /proc/sys/net/ipv4/conf/eth0/send_redirects

IPVSADM='/sbin/ipvsadm'
$IPVSADM -C
$IPVSADM -A -t 192.168.1.5:80 -s rr
$IPVSADM -a -t 192.168.1.5:80 -r 192.168.109.129:80 -m -w 1
$IPVSADM -a -t 192.168.1.5:80 -r 192.168.109.130:80 -m -w 1
```

最后在浏览器里访问 192.168.1.5，并不断刷新网页，就可以依次看到“Server1”和“Server2”了，这表明 LVS 在正常地进行负载均衡服务。

从上面的例子也可以看出，在 NAT 模式下，所有真实服务器的默认网关改为了 LVS 负载均衡器，这让 LVS 成为了一个潜在的瓶颈，当网络流量达到 LVS 服务器的物理限制时，整个系统就无法承接更多的请求了，这种情况下 DR 模式就能很好地解决这个问题。依然使用之前的三台服务器做演示，但是这时候三台服务器的网络将调整到同一个网段下，这里笔者所使用三台服务器的 IP 分别为 192.168.1.5、192.168.1.6、192.168.1.7，虚拟访问 IP 为 192.168.1.10。

在 LVS 服务器上，运行如下脚本：

```
#!/bin/bash
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects
echo 0 > /proc/sys/net/ipv4/conf/default/send_redirects
echo 0 > /proc/sys/net/ipv4/conf/eth0/send_redirects

/sbin/ifconfig eth0:0 192.168.1.10 broadcast 192.168.1.10 netmask 255.255.255.255
up
/sbin/route add -host 192.168.1.10 dev eth0:0

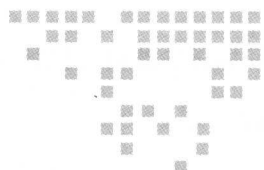
IPVSADM='/sbin/ipvsadm'
$IPVSADM -C
$IPVSADM -A -t 192.168.1.10:80 -s rr
```

```
$IPVSADM -a -t 192.168.1.10:80 -r 192.168.1.6:80 -g -w 1  
$IPVSADM -a -t 192.168.1.10:80 -r 192.168.1.7:80 -g -w 1
```

在真实服务器上，运行如下脚本：

```
#!/bin/bash  
echo "1" > /proc/sys/net/ipv4/conf/lo/arp_ignore  
echo "2" > /proc/sys/net/ipv4/conf/lo/arp_announce  
echo "1" > /proc/sys/net/ipv4/conf/all/arp_ignore  
echo "2" > /proc/sys/net/ipv4/conf/all/arp_announce  
sysctl -p > /dev/null 2>&1  
  
ifconfig lo:0 192.168.1.10 netmask 255.255.255.255 broadcast 192.168.1.10  
route add -host 192.168.1.10 dev lo:0
```

在浏览器里访问 192.168.1.10，并不断刷新，若依次显示为“Server1”和“Server2”，则配置成功。



Graphite

7.1 Graphite 是什么

7.1.1 Graphite 不是一个告警系统

提到监控，大部分系统管理员想到的都是像 Nagios、Zabbix 这样的告警系统，它们会在系统或软件发生异常的时候实时地为系统管理员提供告警信息。事实上，实时的告警只是监控的一个部分，除此之外，为管理员和开发人员提供软件实时的性能指标（metrics）查询、提供历史数据对照等都属于监控的重要组成部分。

Graphite 就是这样一个存储和展现性能指标的优秀开源软件。Graphite 项目于 2006 年由 Orbitz.com 创建，并于 2008 年开源，在 <https://github.com/graphite-project> 上可以找到它的源代码。

7.1.2 Graphite 的功能和特色

Graphite 的基本功能是接收性能数据，并将其展现成为随时间演化的图像，比如图 7-1 展示的是 12 小时内服务器负载均值（load average）的监控数据。

图 7-1 中的横轴为时间，纵轴为数值，下方的字符串代表 3 个不同的性能指标，在 Graphite 中称之为 metrics，图片展示的便是对应 metrics 在 12 小时内的变化。

注意，Graphite 自己本身并不收集各种性能指标，而是需要我们将性能指标发送给它，不过，给 Graphite 发送数据其实是非常简单的，只是需要把 metrics 名、时间戳和对应数据值发送给服务即可（后面可以看到）。

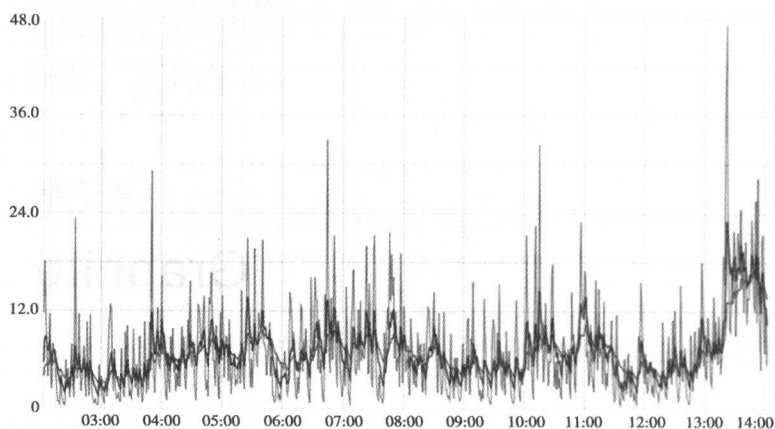


图 7-1 12 小时内负载均衡指标

能画图并不能说明 Graphite 为何深受系统管理员和开发人员的欢迎，毕竟像 Zabbix、Nagios、Cacti 这样的软件其自身或者插件也都提供了类似功能。那么，我们就来看看 Graphite 还有哪些特色：

- ❑ 数据实时展现。这是 Graphite 最大的亮点之一，发送给 Graphite 的数据可以实时地在它的 Web 页面上展现出来。从服务器发送到展现在 Graphite Web 页面的延时基本可以忽略不计。
- ❑ 高度可扩展。通过添加物理硬件，Graphite 可以很好地实现性能上的平行扩充。
- ❑ 丰富的作图功能。Graphite 提供了丰富作图函数，可以对数据进行各种绘制操作，比如对数据进行求和、求差值、合并图像等操作。
- ❑ 简单易用的 API。向 Graphite 请求数据的 API 非常简单，通过在 HTTP url 中提供各种参数，便可以轻松获得我们想要的数据库。

7.2 Graphite 的基本组件

Graphite 软件由 Graphite WebUI、Carbon 和 Whisper 三个组件组成，其中，Graphite WebUI 用来展现数据；Carbon 用来接收数据；Whisper 用来存储数据。下面分别简单介绍这三个组件的功能。

7.2.1 Whisper

Whisper 是 Graphite 用来存放数据的组件，它是一个大小固定的数据库，类似于 RRD 文件，一个 metrics 存放在一个 Whisper 文件中，一般文件名以 .wsp 结尾。每个 metrics 都有它对应的精度（precision）和保留期（retention），当这两个参数设定好了之后，whisper

文件大小也就随之确定了。

精度和保留期主要用于设置对应数据需要保留多长时间，以及多长时间保存一个数据点。whisper 文件支持灵活的数据保存设置，比如：

```
1m:7d,5m:30d,15m:1y
```

表示最近 7 天内，每分钟保存一个数据点；一个月內，每 5 分钟保存一个数据点；一年内，每 15 分钟保存一个数据点；最多保存一年的数据。这种稀疏化的保存方式，既保证了近期数据精确性，还保证在尽量少使用磁盘的情况下，保留最长时间的数据。

数据保存的时间标志有：

- ☐ s: 表示秒
- ☐ m: 表示分钟
- ☐ h: 表示小时
- ☐ d: 表示天
- ☐ y: 表示年

数据是否需要稀疏化保存可以随意设置，比如只设置 1m:7d 也是可以的。除了使用时间来设置精度和保留期之外，还可以通过“每个数据点的秒数:数据点个数”的形式来设置，比如：

```
60:1440
```

表示每个数据点为一分钟，也就是一分钟保存一个数据，一共保存 1440 个数据点，也就是一天的数据。

具体设置 precision 和 retention 的配置文件叫做 storage-schema.conf，在后续章节里会介绍到。数据的稀疏化过程都是 Graphite 自动处理的，并不需任何的人为干预。

7.2.2 Carbon

Carbon 是 Graphite 中接收数据的组件，Carbon 根据功能的不同有三个不同的守护进程，具体如下。

1. Carbon Cache

Carbon Cache 用来接受发送过来的 metrics，它将收到的数据缓存在内存中，同时批量写入 Whisper 数据库。数据发送可以使用多种 metrics 发送协议，比如明文发送和使用 python pickle 格式。

2. Carbon Relay

Carbon Relay 主要用来做两件事情：数据复制和数据分片。数据复制是指将同一个数据发送到不同的目的地，保证数据的高可用性；数据分片是指将不同的数据发送到不同的目的地，保证数据接收端的可扩展性，这样一来，就不会由于大量的数据而造成性

能问题了。

一般来说，Carbon Relay 配置在 Carbon Cache 之前。脚本在将数据发送给 Carbon Relay 后，Relay 会将数据转发给 Carbon Cache，Carbon Cache 则将数据保存到磁盘。

3. Carbon Aggregate

主要用来对 metrics 进行聚合，它将 metrics 缓存起来，在一段时间之后对其进行聚合操作，比如求和或者求平均值，再将其发送给 Carbon Cache。

7.2.3 Graphite Web

Graphite Web 是用来展现 metrics 的一个 Web 界面，使用 Python Web 框架 Django 写成。其数据来源是 Whisper 数据库文件和保存在 Carbon Cache 内存里的 metrics。Graphite 的实时性就来源于此，数据不需要写入到磁盘就可以被 Web 页面读取，从而避免了由于数据写入磁盘而产生的延迟，数据从产生到显示在 Web 界面之间的延时基本上就是数据在网络上传输所需要的时间。图 7-2 是一个 Graphite Web 的截图。

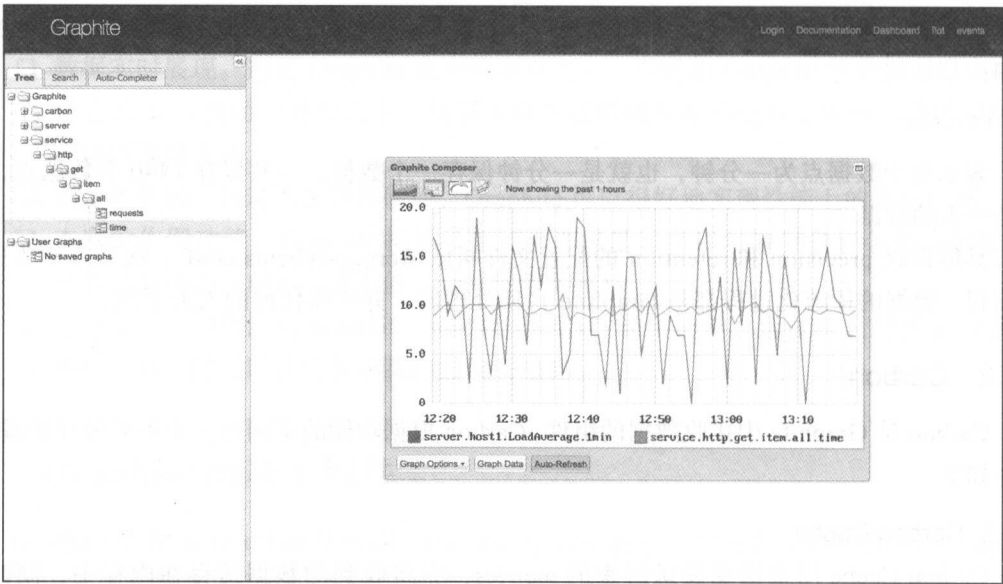


图 7-2 Graphite WebUI 示意图

7.3 Graphite 的安装

下面分开讲述 Graphite 各个组件的安装配置过程，这里涉及的所有源代码都是从 GitHub 上拉取的。关于 GitHub 的使用，请读者自行参考相关文档。

7.3.1 安装 Whisper 数据库

在安装 Whisper 数据库之前，要从 GitHub 获取 Whisper 数据库的源代码：

```
git clone git@github.com:graphite-project/whisper.git
```

然后就可以使用 Python 从源代码安装 Whisper 了，安装代码如下：

```
cd whisper && sudo python setup.py install
```

以上命令会在服务器上安装以下文件。Whisper 数据库不需要特殊的配置，它是 Carbon 和 Graphite-web 需要调用的 Python 库。

```
/usr/bin/rrd2whisper.py
/usr/bin/whisper-create.py
/usr/bin/whisper-dump.py
/usr/bin/whisper-fetch.py
/usr/bin/whisper-info.py
/usr/bin/whisper-merge.py
/usr/bin/whisper-resize.py
/usr/bin/whisper-set-aggregation-method.py
/usr/bin/whisper-update.py
/usr/lib/python2.6/site-packages/whisper-0.9.12-py2.6.egg-info
/usr/lib/python2.6/site-packages/whisper.py
/usr/lib/python2.6/site-packages/whisper.pyc
```

7.3.2 安装 Carbon 守护进程

安装 Carbon 守护进程的步骤如下。

1) 从 GitHub 获取 Carbon 源代码，如下：

```
git clone git@github.com:graphite-project/carbon.git
```

2) 在 Python 上根据源代码安装 Carbon，命令如下：

```
cd carbon && sudo python setup.py install
```

3) Carbon 需要用到 Python 的 twisted 库，一般发行版都自带了 twisted 的包，在 CentOS 中，可以通过 yum 安装，命令如下：

```
yum install -y python-twisted
```

经过上述步骤，Carbon 的文件被安装到了 /opt/graphite 目录，它的守护进程文件包括：

```
/opt/graphite/bin/carbon-aggregator.py
/opt/graphite/bin/carbon-cache.py
/opt/graphite/bin/carbon-client.py
/opt/graphite/bin/carbon-relay.py
```

7.3.3 安装 graphite-web

安装 graphite-web 的步骤如下。

1) 从 github 获取源代码, 如下:

```
git clone git@github.com:graphite-project/graphite-web.git
```

2) 从源代码安装 Graphite, 命令如下:

```
cd graphite && sudo python setup.py install
```

3) 安装依赖包。Graphite 依赖于一系列的 Python 库, 这些库基本上 CentOS 都自带了, 可以通过 yum 直接安装。可通过下面命令来检查所有的依赖关系是否都已经满足:

```
[root@graphite01 graphite]# yum install -y Django django-tagging pycairo python-ldap python-memcached python-twisted python-simplejson python-txamqp python-cairocffi pyparsing pytz bitmap-fonts
```

```
[root@graphite01 graphite]# python check-dependencies.py
```

```
[OPTIONAL] Unable to import the 'python-rrdtool' module, this is required for reading RRD.
```

```
1 optional dependencies not met. Please consider the optional items before proceeding.
```

```
All necessary dependencies are met.
```

7.4 Graphite 的配置 (单点)

本节将讲述如何配置一个单点的 Graphite 服务。在该节中, 是将分析数据由一个简单的脚本发送给 Carbon-Cache 守护进程, 写到本机的 Whisper 数据库, 然后 graphite-web 读取并显示数据的过程。

7.4.1 配置 Carbon 守护进程

启动 Carbon 守护进程, 需要先读取它的配置文件, 配置文件位于 /opt/graphite/conf/carbon.conf 下。笔者使用的配置文件如下:

```
[cache:1]
LINE_RECEIVER_INTERFACE = 0.0.0.0
LINE_RECEIVER_PORT = 2003
PICKLE_RECEIVER_INTERFACE = 0.0.0.0
PICKLE_RECEIVER_PORT = 2004
CACHE_QUERY_INTERFACE = 0.0.0.0
CACHE_QUERY_PORT = 7201
MAX_CACHE_SIZE = inf
MAX_UPDATES_PER_SECOND = 250
MAX_UPDATES_PER_SECOND_ON_SHUTDOWN = 500
WHISPER_AUTOFLUSH = False
LOG_UPDATES = True
```

注意, 在安装了 Carbon 之后, 会有个配置文件样例 /opt/graphite/conf/carbon.conf.example, 其中有对每个选项的详细解释, 读者也可以将这个文件直接拷贝为 carbon.conf。这里建议

使用本书中的样例。

对于上面的配置文件，各个选项的具体解释如下：

- ❑ `LINE_RECEIVER_INTERFACE`、`LINE_RECEIVER_PORT`、`PICKLE_RECEIVER_INTERFACE`、`PICKLE_RECEIVER_PORT`：这些选项用于配置不同数据接收协议监听的网卡和端口。之前也提到过，Carbon 可以接受两种数据格式：一种是明文的字符串，一种是 Python 的 PICKLE 数据格式，分别对应这里的 `LINE_RECEIVER` 和 `PICKLE_RECEIVER`。INTERFACE 一般配置为 0.0.0.0，表示监听所有的网卡，端口在这里分别配置为 2003 和 2004，使用的是 TCP 协议。
- ❑ `CACHE_QUERY_INTERFACE`、`CACHE_QUERY_PORT`：Graphite Web 除了从 Whisper 数据库里读取数据外，还可以从 Carbon Cache 进程读取缓存在内存里面的数据，这时候需要配置一个服务端口给 Graphite Web，这就是 `CACHE_QUERY` 这两个选项的作用，用于配置监听网卡和端口。
- ❑ `MAX_CACHE_SIZE`：这个选项来控制 Carbon Cache 守护进程可以缓存在内存中的数据大小。一般设置为 inf，表示无限制。如果缓存的数据过大，会导致服务器使用 swap，并且 Carbon Cache 需要对缓存的数据进行排序等操作，可能会对 CPU 的使用造成瓶颈。但是缓存数据过大一般是由于内存写入到磁盘不够快而造成的，所以从优化的角度来讲，还是需要解决写入的速度问题。
- ❑ `MAX_UPDATES_PER_SECOND`、`MAX_UPDATES_PER_SECOND_ON_SHUTDOWN`：这两个选项设置在平时和关闭 Carbon Cache 的时候，每分钟写入到 Whisper 数据库操作的次数。主要用来防止写入磁盘过快而导致磁盘问题。`MAX_UPDATES_PER_SECOND_ON_SHUTDOWN` 主要是考虑到关闭 Carbon Cache 的时候，必须将内存里面的数据迅速写入到磁盘，这样 Carbon Cache 才能尽快地关闭，所以一般这个值的设置都比 `MAX_UPDATES_PER_SECOND` 稍大。
- ❑ `WHISPER_AUTOFLUSH`：这个选项表示跳过 kernel buffer 而直接写入硬盘。一般来说不要这么做，通常设置为 False。
- ❑ `LOG_UPDATES`：默认情况下，Carbon Cache 将每次 Whisper 的更新操作都写入 log 文件里面，当 metrics 很多的时候，更新操作是非常频繁的，如果 log 文件和 Whisper 是同时保存在同一个磁盘卷下面的，那么这些操作将有可能影响 Whisper 的写入性能，所以一般将这个选项设置为 False。但是对于本书的演示来说，没有太高的性能要求，在这里设置为 True，方便通过查看 log 监控 whisper 操作。

可能有读者已注意到配置文件里有个 `[cache:1]`，而 `carbon.conf.example` 文档里面的 `[carbon]`。这是因为 carbon-cache 可以启动多个进程，对不同的进程，需要有不同的配置，比如监听端口等。而这里正好配置了一个 carbon-cache 实例 1 的进程，因此在配置文件里的为 `[cache:1]`。通过下面的命令：

```
python /opt/graphite/bin/carbon-cache.py --instance 1 start
```

可将 Carbon Cache 守护进程启动起来。默认的情况下，如果不加 instance 参数，Carbon 会启动一个名叫 a 的进程。

通过 netstat 命令可以看到所有配置的端口都已经被监听了，如下：

```
[root@graphite01 graphite]# netstat -ntlp |grep python
tcp        0      0 0.0.0.0:2003          0.0.0.0:*            LISTEN     1927/python
tcp        0      0 0.0.0.0:2004          0.0.0.0:*            LISTEN     1927/python
tcp        0      0 0.0.0.0:7201          0.0.0.0:*            LISTEN     1927/python
```

配置完 Carbon Cache 之后，需要配置 /opt/graphite/conf/storage-schemas.conf，告诉 Whisper 将以什么频率来发送 metrics，以及希望数据保留多长时间。配置命令如下：

```
cat /opt/graphite/conf/storage-schemas.conf
[carbon]
pattern = ^carbon\.
retentions = 60:90d

[default]
pattern = .*
retentions = 60s:7d,5m:30d,15m,1y
```

这个文件将不同的 metrics 保留期设置为了不同的部分，这里是 [carbon] 和 [default] 两个部分，这两个名字可以随意设置，一般使用一个具有代表意义的名字即可。其中所包含的两个配置又分别为：

- ❑ **Pattern**：用正则表达式设置一个模式匹配的规则，metrics 的名字如果匹配这个正则，就使用此部分设置的保留期。
- ❑ **Retention**：设置保留期，Retention 的设置格式在前文已经有详细讲述。这里的意思是 7 天内的数据，60 秒保存一个数据点；30 天内的数据，5 分钟保存一个数据点；一年内的数据，15 分钟保存一个数据点；多于一年的数据丢弃。

7.4.2 给 Carbon Cache 发送数据

首先使用一个脚本产生一些简单的数据，脚本代码如下：

```
#!/usr/bin/env bash
for i in `seq 1 20`;
do
    echo server.host${i}.LoadAverage.15min $((($RANDOM%20)) $(date +%s))
    echo server.host${i}.LoadAverage.5min $((($RANDOM%20)) $(date +%s))
    echo server.host${i}.LoadAverage.1min $((($RANDOM%20)) $(date +%s))
done
```

然后，将以上代码保存为 /root/metrics_gen.sh。运行 bash /root/metrics_gen.sh，可以看到类似如下输出：

```
server.host20.LoadAverage.5min 16 1421330774
server.host20.LoadAverage.1min 0 1421330774
```

这就是 Carbon 接受的明文数据格式。上面一行数据根据空格可以分割为 3 个部分，第一个部分 `server.host20.LoadAverage.5min`，这是 `metrics` 的名字。Carbon 会在 Whisper 的根目录（一般为 `/opt/graphite/storage/whisper/`）下，按照 Whisper 的名字，以点号作为分隔符，自动生成 `server/host20/LoadAverage` 这个目录，并把数据保存到 `5min.wsp` 这个文件中。其中，16 是这个 `metrics` 的值，而 1421330774 是对应数据生产的时间戳。

现在使用下面这个命令向 Carbon Cache 发送数据：

```
bash metrics_gen.sh | nc localhost 2003
```

这里只是简单地将产生的字符串通过 `nc` 命令发送给了 Carbon Cache 监听的 2003 端口。此时查看 `/opt/graphite/storage/whisper/server/`，便可以看到自动生成的 `host` 目录以及里面的 `whisper` 文件。

```
[root@graphite01 server]# ls -lh /opt/graphite/storage/whisper/server/host7/
LoadAverage/*
-rw-r--r--. 1 root root 631K Jan 15 22:17 /opt/graphite/storage/whisper/server/
host7/LoadAverage/15min.wsp
-rw-r--r--. 1 root root 631K Jan 15 22:17 /opt/graphite/storage/whisper/server/
host7/LoadAverage/1min.wsp
-rw-r--r--. 1 root root 631K Jan 15 22:17 /opt/graphite/storage/whisper/server/
host7/LoadAverage/5min.wsp
```

而且可以看到这些文件的大小都是一样的，对于一个 Whisper 文件，如果保留期确定之后，它的大小也就确定了，不会随时间的变化而增长。

接下来配置一个 `cronjob`，保证数据能够持续地发送到 Carbon Cache。命令如下：

```
[root@graphite01 ~]# cat /etc/cron.d/metrics_sender
* * * * * root bash /root/metrics_gen.sh | nc localhost 2003
```

这里设置了每分钟运行一次这个脚本，并将 `metrics` 发送给 Carbon Cache。

Carbon Cache 的日志保存在 `/opt/graphite/storage/log/carbon-cache` 下面，每一个 instance 有一个目录来保存它们的日志。进入 `carbon-cache-1` 目录，可以看到有如下日志文件产生：

```
[root@graphite01 carbon-cache-1]# ls
console.log creates.log listener.log query.log updates.log
```

7.4.3 配置 Graphite-web

现在来配置 Graphite-web。首先要在服务器上安装好 MySQL，MySQL 主要用来保存配置好的 Graphite Dashbord 数据。其次还需安装 Apache 和 `mod_wsgi`，用它来运行 Graphite 的 Web 服务。此外，还需使用到 Python 的 MySQL 库，Graphite 会用它来连接 MySQL。安装 MySQL 的命令如下：

```
yum install -y mysql mysql-server httpd MySQL-python mod_wsgi
```

启动 MySQL 和 Apache 的命令如下：

```
/etc/init.d/mysqld start && /etc/init.d/httpd start
```

现在，打开 MySQL shell，创建 Graphite 的数据库用户，命令如下：

```
[root@graphite01 graphite]# mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
mysql> grant all on graphite.* to 'graphite'@'localhost' identified by "graphite";
Query OK, 0 rows affected (0.00 sec)
```

此时需要配置 Graphite-web 本身的一些参数，配置文件在 /opt/graphite/webapp/graphite/settings.py 下，一般不改动这个文件，而是创建一个 local_settings.py，Graphite 的安装里面自带了一个 local_settings.py.example 文件，里面有各个选项的详细解释，笔者使用的 local_settings.py 内容如下：

```
[root@graphite01 graphite]# cat local_settings.py
DATABASES = {
    'default': {
        'NAME': 'graphite',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'graphite',
        'PASSWORD': 'graphite',
        'HOST': 'localhost',
        'PORT': '3306'
    }
}

CARBONLINK_HOSTS = ['localhost:7201:1']
TIME_ZONE = 'UTC'
```

其中，Database 选项表示我们使用 MySQL 作为后端数据库，并且还配置了连接数据库的各个参数，比如用户名、密码、主机、端口等。

此外，CARBONLINK_HOSTS 也是一个比较重要的参数，graphite-web 可以读取 Carbon Cache 保存在内存里还没写到磁盘中的 metrics，但是必须告知 graphite-web 从哪里读取这些数据，即通过指定 CARBONLINK_HOST 来设置。之前 Carbon Cache 里设置过 CACHE_QUERY_INTERFACE 和 CACHE_QUERY_PORT，在这里，需要将 CARBONLINK_HOST 设置为相对应的主机和端口。若出现不匹配的情况，Graphite WebUI 将无法正确地 Carbon Cache 中读取数据。因为这里将 graphite-web 和 Carbon Cache 安装在同一个服务器上面，所以这里设置为 localhost 即可，端口是对应的 CACHE_QUERY_PORT，结尾的“:1”必须和前面 Carbon Cache 的命名一致。CARBONLINK_HOSTS 可以设置多个值，从多个 Carbon Cache 读取保存在内存里面的 metrics。

接下来初始化数据库，运行下面的命令即可完成：

```
python /opt/graphite/webapp/graphite/manage.py syncdb
```

最后配置 Graphite 运行在 Apache 下。首先，要创建 /opt/graphite/conf/graphite.wsgi 文件，文件内容如下：

```
import os, sys
sys.path.append('/opt/graphite/webapp')
os.environ['DJANGO_SETTINGS_MODULE'] = 'graphite.settings'
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
# READ THIS
# Initializing the search index can be very expensive, please include
# the WSGIScriptImport directive pointing to this script in your vhost
# config to ensure the index is preloaded before any requests are handed
# to the process.
from graphite.logger import log
log.info("graphite.wsgi - pid %d - reloading search index" % os.getpid())
import graphite.metrics.search
```

然后，配置 Apache 下的 Graphite，编辑配置文件 /etc/httpd/conf.d/graphite.conf，其内容如下：

```
WSGISocketPrefix run/wsgi
WSGIDaemonProcess graphite processes=5 threads=5 display-name='%{GROUP}'
    inactivity-timeout=120
WSGIProcessGroup graphite
WSGIApplicationGroup %{GLOBAL}
WSGIImportScript /opt/graphite/conf/graphite.wsgi process-group=graphite
    application-group=%{GLOBAL}
WSGIScriptAlias / /opt/graphite/conf/graphite.wsgi
```

更改 graphite-web 文件的权限，保证 Apache 能够访问，如果开启了 SELinux，还需要先关闭 SELinux，命令如下：

```
chown -R apache.apache /opt/graphite/{storage,webapp}
```

重启 httpd，命令如下：

```
/etc/init.d/httpd restart
```

这样 Graphite 就启动起来了，访问本机 http://localhost/，就能看到 Graphite 的 Web 界面。

7.5 Graphite 的配置（集群配置）

根据前一节的讲解，相信读者对 Graphite 的基础组件有了一定的了解。前面使用了一个 Carbon Cache 进程来接收数据并写入磁盘，现在来考虑 Graphite 的扩容问题。假设随着业务数据量的增多，单进程的 Carbon Cache 成为了服务的瓶颈，它无法迅速地处理海量

的 metrics 请求，也无法迅速地将数据写入到磁盘中。值得注意的一点是，虽然在磁盘无故障的情况下，无法迅速地写入磁盘并不是很大的问题，因为 *graphite-web* 可以将数据从内存里读取出来，但是大量的数据，会导致 *graphite-web* 从 Carbon Cache 中读取数据的速度变慢。

面对这样的情况，该如何处理呢？此时很自然的解决方法是添加额外的 Carbon Cache 进程，让它们监听不同的端口，在这些进程前面放置一个 VIP，数据先发送到这个 VIP，然后转发到不同的进程端口。这时候必须要保证对应一个 metrics，它永远只能发送到同一个 Carbon Cache 进程，因为如果发送到不同的 Carbon Cache，那么会产生一种“不同的 Cache 进程在同一时刻写同一个 Whisper 文件”的可能性，这有可能会造成文件写入冲突，产生不可预料的文件损坏，或者会使进程在等待其他进程时释放写入锁，从而导致写入的性能大大下降。

添加多个 Carbon Cache 进程的同时，需要将这些进程的信息配置到 *graphite-web* 的 CARBONLINK_HOSTS 选项中，以便 *graphite-web* 能够读取所有的缓存数据。*graphite-web* 读取某一个 metrics 的数据的时候，它最好能够知道这个 metrics 能从哪个 Carbon Cache 进程中读取，这样它就没有必要轮询所有的 Cache 进程了。

因此要做到 Graphite 的集群化，主要有两点需求：

- ❑ 同一 metrics 必须发送到同一个 Carbon Cache 实例。
- ❑ 对于一个 metrics，Graphite WebUI 必须知道从哪个 Carbon Cache 中读取其数据。

这样的需求是通用的负载均衡软件或者硬件不能提供的，幸运的是 Graphite 的开发人员开发出了 Carbon Relay，可以完美地解决这些问题。

7.5.1 配置 Carbon Relay

首先要在前面案例的基础上再启用一个 Carbon Cache 进程。可在 `/opt/graphite/conf/carbon.conf` 下添加如下配置，启动第二个 Cache 进程。

```
[cache:2]
LINE_RECEIVER_INTERFACE = 0.0.0.0
LINE_RECEIVER_PORT = 2103
PICKLE_RECEIVER_INTERFACE = 0.0.0.0
PICKLE_RECEIVER_PORT = 2104
CACHE_QUERY_INTERFACE = 0.0.0.0
CACHE_QUERY_PORT = 7202
MAX_CACHE_SIZE = inf
MAX_UPDATES_PER_SECOND = 250
MAX_UPDATES_PER_SECOND_ON_SHUTDOWN = 500
WHISPER_AUTOFLUSH = False
LOG_UPDATES = True
/opt/graphite/bin/carbon-cache.py --instance 2 start
```

然后在 `/opt/graphite/conf/carbon.conf` 中为 Carbon Relay 添加如下配置：

```
[relay:1]
RELAY_METHOD = consistent-hashing
LINE_RECEIVER_INTERFACE = 0.0.0.0
LINE_RECEIVER_PORT = 2013
PICKLE_RECEIVER_INTERFACE = 0.0.0.0
PICKLE_RECEIVER_PORT = 2014
MAX_QUEUE_SIZE = 10000
MAX_DATAPOINTS_PER_MESSAGE = 500
USE_FLOW_CONTROL = True
DESTINATIONS = localhost:2004:1,localhost:2104:2
```

对于上面的配置文件，各个选项的具体解释如下。

- ❑ **RELAY_METHOD**：这个选项用于设置 Carbon Relay 转发 metrics 的方式。转发方式分别有如下几种。
 - **rules**：表示在 relay-rules.conf 里面配置匹配模式，不同的匹配发送到不同的目的地，Relay 的目的地可以是 Cache，也可以是 Relay，还可以是 Aggregator。
 - **consistent-hash**：指 metrics 平均地转发到不同的目的地。Carbon Relay 使用一套内部机制，保证同一 metrics 永远发送到特定的目的地。
 - **aggregated-consistent-hashing**：当 Carbon Relay 的转发数据的后端是 Carbon Aggregator 的时候使用。
- ❑ **LINE_RECEIVER_INTERFACE、LINE_RECEIVER_PORT、PICKLE_RECEIVER_INTERFACE 和 PICKLE_RECEIVER_PORT**：这几个选项和 Carbon Cache 中的对应选项意思相同，即配置不同数据格式的监听地址。
- ❑ **MAX_QUEUE_SIZE**：Carbon Relay 中保持 metrics 队列的长度。
- ❑ **USE_FLOW_CONTROL**：如果设置为 False，Carbon Relay 的任意发送队列超过 MAX_QUEUE_SIZE 的时候，Relay 会开始丢弃保存的数据；如果设置为 True（默认值），接收 metrics 的那个 socket 会停止接收新数据，直到发送队列小于 MAX_QUEUE_SIZE * 0.8。
- ❑ **MAX_DATAPOINTS_PER_MESSAGE**：每次发送到后端 Message 的最大数据量。
- ❑ **DESTINATIONS**：发送的目的地，每一个目的地是一个 Carbon 守护进程的实例。注意 Carbon 守护进程之间发送数据使用 PICKLE 格式，因此在这里需要使用 PICKLE_RECEIVER_PORT。

除此之外，还需要更新一个非常重要的参数，即 webapp local_settings.py 里的 CARBONLINK_HOSTS。打开这个文件，将新的 Carbon Cache 添加进去。

```
CARBONLINK_HOSTS = ['localhost:7201:1', 'localhost:7202:1']
```

这个选项需要更加详细的解释。Carbon Relay 使用一套内部算法来计算某个 metrics 应该发送到哪个目的地，而 Graphite Web 在读取这个 metrics 的时候使用的是同一算法，它从 Web 界面上获得用户需要查询的 metrics，从对应的 CARBONLINK_HOST 中读取缓存在内

存里的数据，但它并不会轮询所有的 `CARBONLINK_HOST`。因此在这里这个选项里面的值需要与 `Relay` 的 `DESTINATIONS` 一一对应，顺序也必须完全一致，不然 `Graphite Web` 会读取错误的 `Carbon Cache` 但是取不到任何数据，从而造成缓存在内存中的 `metrics` 无法及时显示，那么就不是一个 `real-time` 的 `Graphite` 服务了。

现在可以启动 `Relay` 进程了，命令如下：

```
python /opt/graphite/bin/carbon-relay.py --instance 1 start
```

然后修改 `Cron`，将 `metrics` 发送到 `Carbon Relay`：

```
cat /etc/cron.d/metrics_sender
* * * * * root bash /root/metrics_gen.sh | nc localhost 2013
```

此时，查看 `Relay` 的日志，可以看到守护进程接收 `metrics` 的记录，如下：

```
tail -2 /opt/graphite/storage/log/carbon-relay/carbon-relay-1/listener.log
19/01/2015 21:55:01 :: MetricLineReceiver connection with 127.0.0.1:56882
    established
19/01/2015 21:55:01 :: MetricLineReceiver connection with 127.0.0.1:56882 closed
    cleanly
```

再次查看 `Graphite Web` 界面，`metrics` 应该仍然会在 `Web` 上正常显示。

7.5.2 Relay 中的数据复制

`Relay` 将不同的 `metrics` 发送到不同的目的端，这个功能叫做数据的分片（`sharding`），除此之外它还有一个功能，叫做数据复制（`replication`），即将同一数据发送到不同的目的端，从而防止数据丢失。

`Relay` 也支持数据复制功能，它通过选项 `carbon.conf` 中的 `REPLICATION_FACTOR` 来制定复制功能。`REPLICATION_FACTOR` 指的是数据发送的份数，默认值为 1，表示只发送一份，如果将 `REPLICATION_FACTOR` 改成 2，那么 `Relay` 会将同一份数据发送到两个不同的目的端。这里有一个有趣的技巧，我们可以创建两个目的地，然后将 `REPLICATION_FACTOR` 设置成 2，这样就有了一个数据的完全备份，两边都有全部数据。笔者在生产环境中就使用了这个技巧，除了正常的 `Graphite` 服务器之外，还使用 `replication` 将数据发送到一个 `Graphite` 缓存服务器，它配备有 `SSD` 磁盘，只用来存放最近一周的数据，由于保存数据少，这台服务器的响应非常快，数据量上也满足绝大部分的日常需求。

7.5.3 数据聚合

什么是数据聚合呢（`aggregation`）？数据聚合典型的应用场景就是统计平均值和计数器。比如服务 `a` 需要时常访问另外一个服务 `b`，我们想统计一分钟内访问 `b` 的次数，或者想要计算访问 `b` 服务的平均响应时间，那么就需要将每一次从 `a` 访问 `b` 的响应时间数据发送出去，而 `metrics` 接收端需要将这些数据暂时缓存起来，对时间计算一个累计值，或者平均值，然

后再进行数据上报。这样一个过程叫做数据聚合。

Carbon Aggregator 就是一个用来聚合数据的服务，它支持两种聚合方式：求和（sum）和求平均（avg）。在 `/opt/graphite/conf/carbon.conf` 里添加如下聚合配置：

```
[aggregator:1]
LINE_RECEIVER_INTERFACE = 0.0.0.0
LINE_RECEIVER_PORT = 2023
PICKLE_RECEIVER_INTERFACE = 0.0.0.0
PICKLE_RECEIVER_PORT = 2024
DESTINATION_HOST = 127.0.0.1
DESTINATION_PORT = 2004
MAX_QUEUE_SIZE = 10000
MAX_DATAPOINTS_PER_MESSAGE = 500
DESTINATIONS = localhost:2004:1, localhost:2104:2
```

这些选项和之前 Relay 选项的意思基本一致，这里不再详述。启动 aggregator，命令如下：

```
python /opt/graphite/bin/carbon-aggregator.py --instance 1 start
```

下面配置 `/opt/graphite/conf/aggregation-rules.conf`，这个文件的任何改动都能够被自动读取到。

```
service.http.get.item.all.requests (60) = sum service.http.get.item.*.requests
service.http.get.item.all.time (60) = avg service.http.get.item.*.time
```

`aggregation-rules.conf` 里的配置格式为 `output_template (frequency) = method input_pattern`。`output_template` 为输出的 metrics 格式，`input_pattern` 表示用来匹配 metrics 的表达式，`frequency` 和 `method` 表示多长时间将 metrics 用何种聚合方式求值并发送出去。

现在用一个脚本发送一些 metrics，编辑 `/root/aggregated_metrics.sh`，命令如下：

```
#!/usr/bin/env bash
count=10
while [ $count -gt 0 ]
do
    for i in $(seq 11 20); do
        echo service.http.get.item.${i}.requests 1 $(date +%s) | nc localhost 2023
        echo service.http.get.item.${i}.time $((($RANDOM % 20)) $(date +%s) | nc localhost
            2023
    done
    sleep $((($RANDOM % 4))
    count=$((count-1))
done
```

把这个脚本安装成为 cron，命令如下：

```
[root@graphite01 ~]# cat /etc/cron.d/metrics_sender
* * * * * root bash /root/metrics_gen.sh | nc localhost 2013
* * * * * root bash /root/aggregated_metrics.sh
```

等待一段时间后，就可以在 Web 界面上看到新的 metrics 了（如图 7-3 所示）。

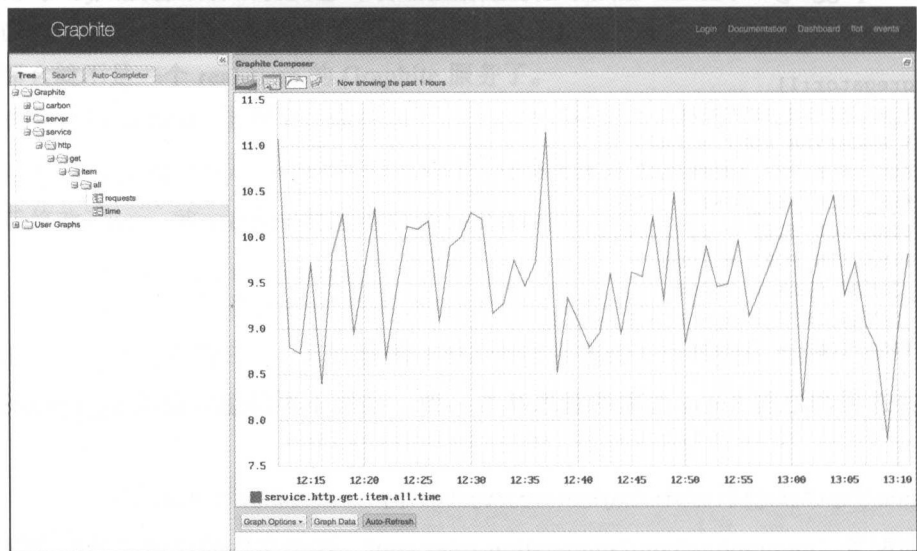


图 7-3 Graphite 聚合 metrics 的示意图

7.5.4 Graphite Cluster

一直到现在，我们的 Graphite 服务都是搭建在一台机器上的。在大型的生产环境中，一台服务器的性能往往不足以支撑接收海量的 metrics，这时就需要增加更多的服务器，搭建一个 Graphite Cluster。

综合前文所述，可以在不同机器上启动多个 Carbon Relay 和 Carbon Cache，然后使用 Carbon Relay 的 consistent-hashing 功能，将 metrics 平均地发送到不同的 Carbon Cache，并保存在不同服务器的磁盘上，从而到达增加性能（capacity）的目的。在 Carbon Relay 之上，需要有一个 VIP，用来做 Relay 的负载均衡，客户机的 metrics 都会发送到此 VIP。

对于 Graphite-web 的配置，需要注意两点：

- ❑ CARBONLINK_HOSTS 必须和 carbon.conf 中 Relay 部分的 destination 完全保持对应，顺序也必须完全一样，这些在前文中已经有相关解释。
- ❑ 由于 metrics 数据保存在不同的服务器中，因此 Graphite Web 需要从其他服务器上读取数据，此时必须配置 CLUSTER_SERVERS 为除本机之外的其他 Graphite 服务器，其值为 [“hostname1:port1, hostname2:port2, …”]。Port 是访问 graphite-web 的 http 端口。有了这个配置，Graphite-web 就能从其他服务器上读取其保存的数据了，这个数据包括硬盘上的 whisper 数据和内存里面的 Carbon Cache 数据。注意，它如果在 hostname1 上找到了需要的数据，就不会再去 hostname2 上找，知道这点

对 metrics 的迁移工作很有必要。也可以将一个存储放置在所有服务器的后端，挂载到 Carbon Cache 服务器上，这样就不需要配置 `CIUSTER_SERVERS` 了。

7.6 使用 Graphite Web

Graphite 的 Web 界面使用起来非常简单。它的首页可以分为两个部分：左边的 metrics 树和右边的图片生成框（Graphite Composer）。metrics 树是一个可点击的层级结构，包含了所有在 Whisper 目录里能找到的 metrics，如果是一个新的 metrics，刚发送到 Carbon Cache，还没来得及保存到磁盘，那么在 metrics 树里面是不会显示的。点击 metrics 树便可以展开并找到自己想要的 metrics，同时在右方的图片生成框中生成相应的图片。

Graphite 的首页上有一些使用技巧，这里简单地提及一下。

- ❑ 点击不同的 metrics，所有数据可以显示在同一图片上。再点击同一 metrics，便可以取消在图片上数据的显示。
- ❑ 图片生成框的右上方可以简单地选择数据显示时间，包括相对时间（前一天，一周等）和绝对时间范围。
- ❑ 右下角的 Graph Options 可以对图片进行各种特殊处理，这是 Graphite Web 深受用户喜爱的一个重要原因，本章会详细讲述 Graphite 作图的一些特殊函数，从而帮助了解其强大的作图功能。

7.6.1 Graphite 的 Render API

在介绍 Graphite 作图函数之前，先来介绍 Graphite 的 Render API，这是 Graphite Web 和其后端数据交换的基础。

首先右键点击 Graphite Composer 上的图片，复制图片的 URL，然后粘贴在浏览器中，应该是下面这种形式的 URL：

`http://localhost/render?width=932&height=564&_salt=1422965430.331&target=carbon.agents.graphite01-2.metricsReceived&from=-2hours`

这个 URL 就是 Graphite 的 Render API，它通过向 http URL 传递不同的参数，控制和调整显示在页面上的图片。

Render 的基本 URL 是 `http://GRAPHITE_HOST:PORT/render`，打开这个链接，默认应该是一张显示为 no data 的图片。这个 URL 接受的一些基本 URL 参数，如下。

- ❑ target 参数：向图片中添加数据的参数即为 target，target 的值是任意 metrics 路径，比如 `target=server.host1.LoadAverage.1min`。
 - target 支持通配符，比如 `target=server.*.LoadAverage.1min`。
 - target 支持基本的扩展，比如 `target=server.host[1-9].LoadAverage.1min`，或者 `target=server.{host1,host2}.LoadAverage.1min`。

○ target 值可以设置多次，比如 `target= server.host1.LoadAverage.1min&target= server.host2.LoadAverage.1min`。

□ from/until 参数：用来设置显示 metrics 的时间范围，支持 from/until=- 相对时间或者 from/until= 绝对时间两种方式。默认的情况下，from 为 -24 小时，而 until 为当前时间，表示显示一天内的数据。

相对时间的单位可以为 s/min/h/d/w/mon/y，分别表示秒、分、时、天、周、月和年，而绝对时间除相对时间支持的单位之外，还可以是 HH:MM_YYMMDD、YYYYMMDD、MM/DD/YY 格式。示例如下：

```
from=-10d&until=-1d
from=20140314&until=20150101
from=Monday
```

□ width/height 参数：这两个参数用来设置图片的大小，单位为像素。比如 `width=932 &height=564`。

□ title 参数：设置图片的标题。

□ format 数：设置显示数据的格式，默认为 PNG 图片。其值可以为 PNG、CSV、SVG 和 JSON 等。

通过调整这些参数，可以获得我们想要的内容，比如图片或者纯数据。大部分 Graphite 的第三方 Dashboard 都是基于 Render 的 API 来编写的。了解和熟悉 Render 的用法对我们灵活使用 Graphite 有很大的帮助。

7.6.2 Graphite 作图函数

针对 Graphite Web 取得的 metrics 数据，还可以通过一些特殊函数进行处理，从而将图片用我们想要的方式展示出来。下面是一些经常会用到的函数。

1. 命名函数

默认的情况下，Graphite 显示在图像上的 metrics 名字为 metrics 的路径，可以通过一系列的函数来进行重命名。

□ Alias 函数：对 metrics 进行重命名，比如 `&target=alias (Sales.widgets.largeBlue," Large Blue Widgets")`

□ aliasByMetric 函数：用 metrics 最后一个字段进行重命名。如 `&target=alias(Sales.widgets.largeBlue)`，此时表示重命名为 largeBlue。

□ aliasByNode 函数：用 metrics 的第几个字段来重命名。如 `&target=aliasByNode (server.host[1-9].LoadAverage.1min,1)`，表示用 host 名来重命名 metrics。字段的序号以 0 开始。

□ aliasSub 函数：用正则表达式进行替换，比如 `&target=aliasSub(server.host[1-9].LoadAverage.1min,"^.*?(host[0-9]*).*$", "\1")`。表示对匹配正则表达式 `^.*?(host[0-9]*).*$`

的 metrics，使用第一个匹配字段进行替换。

2. 计算函数

可以通过一系列的计算函数来对 metrics 的值进行数值计算。

- ❑ `sumSeries`：对一系列的 metrics 值做加法。如 `target=sum(server.host[1-9].LoadAverage.1min)`。
- ❑ `sumSeriesWithWildcards`：在某个位置插入通配符之后再使用 `sumSeries`。如 `&target=sumSeriesWithWildcards(host.cpu-[0-7].cpu-{user,system}.value, 1)`，相当于 `target=sumSeries(host.*.cpu-user.value)&target=sumSeries(host.*.cpu-system.value)`。
- ❑ `diffSeries`：对两个 metrics 做减法，如 `&target=diffSeries(server.host{1,2}.LoadAverage.1min)`。
- ❑ `averageSeries`：计算所有 metrics 的平均值，如 `target=averageSeries(server.*.LoadAverage.1min)`。
- ❑ `integral`：对一个 metrics 的值进行时间上的积分。这个函数主要用于在过去一段时间内对 metrics 所有的值进行求和。如 `target=integral(service.http.get.item.all.requests)`。
- ❑ `derivative`：对时间求导，这个函数和 `integral` 相对应，用来获得 metric 随时间的变化值。
- ❑ `maxSeries`：对所有 metrics，提取它们在同一时刻的最大值，合并成一组新的数据。比如 `maxSeries(server.*.LoadAverage.1min)`。
- ❑ `minSeries`：与 `maxSeries` 类似，提取最小值。

3. 显示函数

- ❑ `averageAbove/averageBelow`：显示平均值在设定值之上/下的 metrics。如 `target=averageAbove(server.*.LoadAverage.1min, 5)`。
- ❑ `grep/exclude`：显示或者排除匹配对应表达式的 metrics。如 `&target=exclude(server.host[1-9].LoadAverage.1min, "host9")`。
- ❑ `highestAverage/highestCurrent/highestMax`：显示平均值、当前值、最大值最高的几个 metrics。比如 `target=highestMax(server.*.LoadAverage.1min, 5)`。
- ❑ `lowestCurrent/lowestAverage`：和 `highestCurrent/highestAverage` 相反。
- ❑ `sortByName`：通过 metrics 名字排序。
- ❑ `sortByMaxima`：通过最大值排序。
- ❑ `sortByMinma`：通过最小值排序。

4. 其他函数

- ❑ `cactiStyle`：像 cacti 一样，图例中显示 metrics 的最大值、最小值和当前值。
- ❑ `consolidateBy`：当图片的像素点少于数据点的个数时，Graphite 默认使用这段时间里的数据平均值，`consolidateBy` 函数可以指定其他的方法，可选的方法有 `sum`、

min、average 和 max。比如 `target=consolidateBy(Sales.widgets.largeBlue, 'sum')`。

☐ stacked: 用堆叠的方式显示数据。

☐ dashed: 用虚线的方式显示数据。

7.6.3 Graphite Dashboard 和 Grafana

Graphite Web 的 Dashboard 应该是 Graphite 中最经常使用的功能之一，它可以用来保存经常访问的一个或多个图像，并且可在同一个页面访问它们。图 7-4 是一个 Dashboard 的截屏。

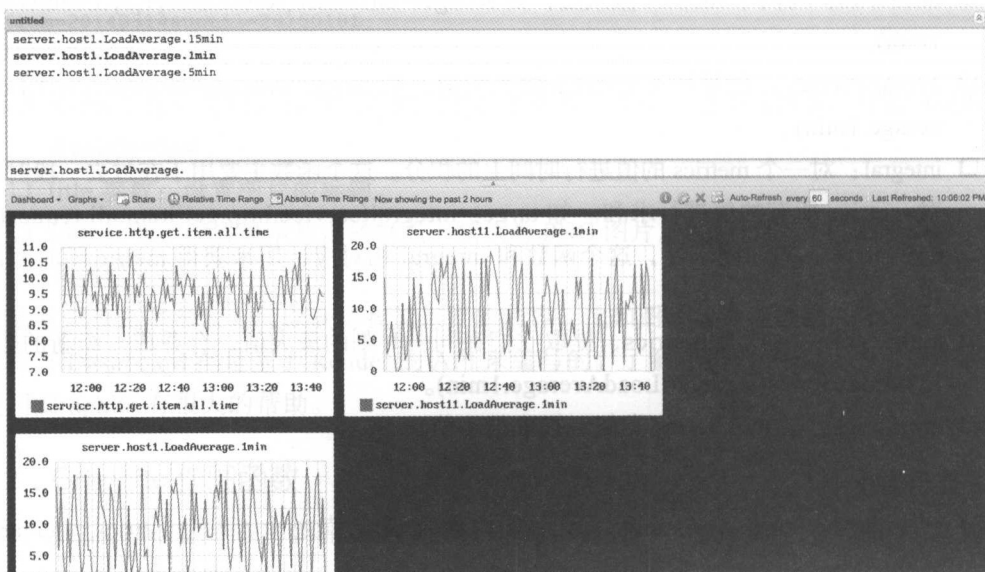


图 7-4 Graphite Dashboard

访问 Dashboard 的 URL 是 `http://Graphite-host/dashboard/`。可以从 Web 上方的 metrics 选择器向 Dashboard 中添加 metrics，之后通过点击 `dashboard → save as` 来保存 Dashboard。之前我们在配置 `graphite-web` 的时候已经配置了 MySQL 数据库，Dashboard 的配置就是保存在数据库里面的。Graphite Dashboard 的使用非常简单，这里不再详述。

总的来说，Graphite 自带的 Dashboard 使用起来其实不是特别方便，主要原因是 Graphite 作图是通过服务器产生图像的，任何更新操作，都需要通过服务器来进行，这样就限制了客户端操作的灵活性。比如想隐藏 Dashboard 中任意的一个 metrics，这样一个简单的操作，应该是一次点击就能够完成的事情，但是对 Dashboard 来说就需要很多次的编辑。

好在 Graphite 有非常多的第三方 Dashboard 实现，在这些软件中，Grafana (`http://grafana.org/`) 是最优秀的实现之一。

Grafana 是一个完全由 JavaScript 写成的 Dashboard 软件 (Grafana 2.0 以后的版本已经不完全是 JavaScript 了), 它的后端可以是 Graphite、openTSDB 等类似的服务。其所有的代码逻辑都是运行在用户的浏览器上面, 作图也是通过 JavaScript 在浏览器上进行的, Graphite 后端只用来提供需要查询的数据。正是由于这一点, Grafana 对在浏览器上的点击、拖曳等操作支持得非常好, 因此使用起来非常方便。下面从安装开始, 简单讲解 Grafana 的使用。

1) 首先从 <http://grafana.org/download/> 下载最新的源代码, 当前版本是 grafana-1.9.1.zip, 解压到 /var/www/html/mv grafana-1.9.1.zip /var/www/html && unzip grafana-1.9.1.zip && mv grafana-1.9.1 grafana。

由于 Grafana 是静态的 js 文件, 此时如果没有安装其他的 http 服务, 通过 <http://hostname/grafana> 就可以访问了。这里因为之前在本机上安装了 Graphite, 所以还需要配置 http。将以下代码保存为 /etc/httpd/conf.d/grafana.conf 并重启 http 服务。

```
Alias /grafana "/var/www/html/grafana"
```

```
<Directory "/var/www/html/grafana">
    Options +Indexes
    AllowOverride All
</Directory>
```

2) 修改 config.js。Grafana 安装包自带一个 config.sample.js, 将它重命名为 config.js, 并将 Graphite 部分的配置文件改为:

```
// Graphite & Elasticsearch example setup
datasources: {
  graphite: {
    type: 'graphite',
    url: "http://GRAPHITE_HOST",
  },
  elasticsearch: {
    type: 'elasticsearch',
    url: "http://my.elastic.server.com:9200",
    index: 'grafana-dash',
    grafanaDB: true,
  }
},
```

在这里需要解释一下相应配置。Grafana 是通过 Graphite 的 render API 获取 metrics 数据并画图的, 因此首先要必须告诉它 Graphite 的服务安装在哪里; 另外, Grafana 也像 Graphite 一样, 可以保存 Dashboard, 不过, 它是使用 Elasticsearch 作为后端的, 关于 Elasticsearch 的相关内容, 可以在本书的其他章节中找到, 这里不加讲述。

此时打开 http://GRAFANA_HOST/grafana/, 就可以看到 Grafana 的首页。图 7-5 是一个默认首页的截图。

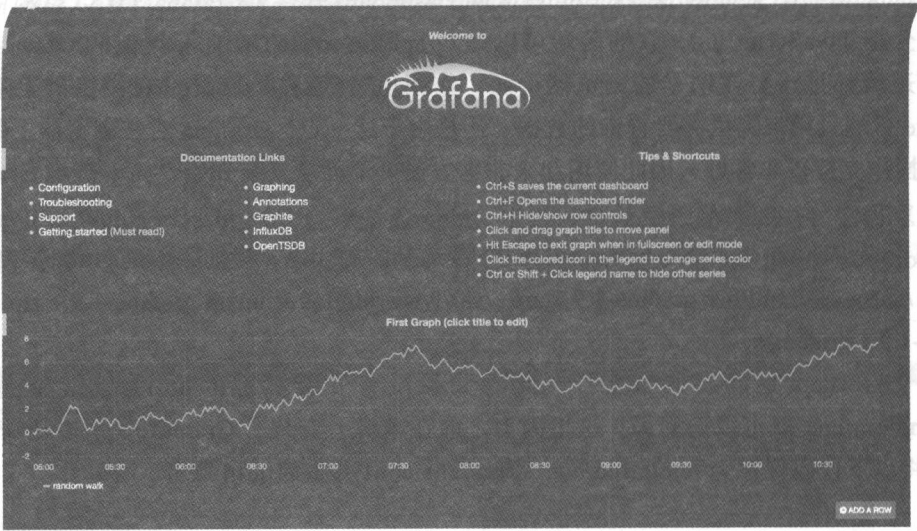


图 7-5 Grafana 首页

Grafana 默认生成了一张图片 First Graph，点击这张图片的标题，然后点击 edit，就可以将这张图片放大并编辑它，如图 7-6 所示。

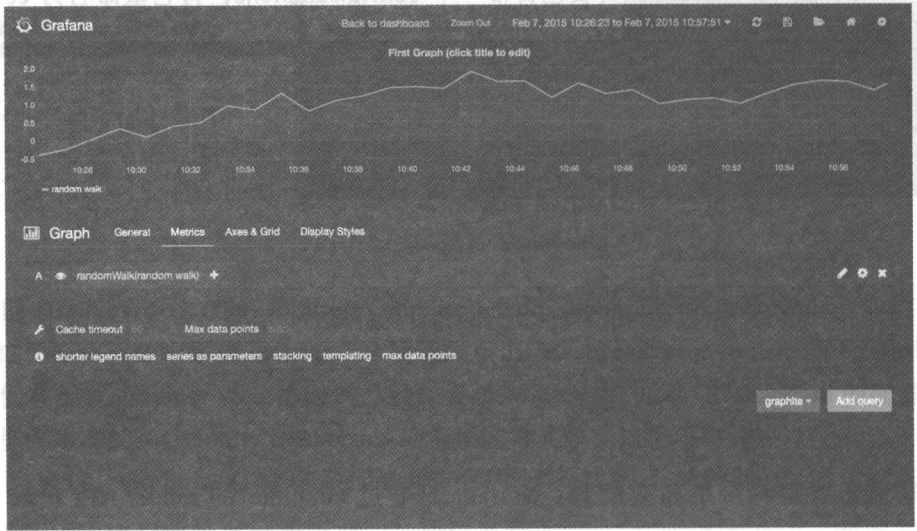


图 7-6 Grafana 图形示例

在这个页面上，因为强大的 JavaScript 和 HTML5，使得我们可以灵活地对这张图像进行编辑操作，比如添加新的 metrics、改变显示方式，等等。在 <http://grafana.org/features/> 可以看到 Grafana 更多有趣的特性，留给读者自行参考。

7.7 Graphite 的性能监控和调整

在理想的情况下，Graphite 的运行状况应该是这样的：metrics 发送到 Relay，Relay 将数据转发给 Carbon，Carbon 迅速将数据写入到 Whisper，供 Graphite Web 读取。这个流程中主要会出现性能问题的地方有如下几个：

- ❑ metrics 无法将尽快地将数据写入到 Whisper 文件，导致 Carbon 服务器过载。

- ❑ Relay 无法将 metrics 尽快地发送到 Carbon，从而出现丢弃数据的情况。

第一个问题主要是磁盘性能的问题，当我们的服务有海量数据时，Whisper 的工作方式决定了它需要磁盘能够支持大规模的小文件读写，此时 SSD 磁盘是支持和解决小文件读写性能问题的最好方案。如果没有 SSD，那么带 RAID 电池的硬盘也是不错的选择。

Carbon 本身有几个配置选项，用来限制 Carbon Cache 过量的使用系统资源，在前文中都有提及。比如 MAX_CACHE_SIZE、MAX_UPDATES_PER_SECOND、MAX_CREATES_PER_MINUTE 和 MAX_UPDATES_PER_SECOND_ON_SHUTDOWN 等。根据笔者的经验，很少会出现需要限制 Carbon Cache 对磁盘操作的情况，大部分性能问题都是磁盘本身性能不足而导致的。比如限制 MAX_UPDATES_PER_SECOND，但是一般出现的问题都是磁盘写入太慢，实际每秒写入磁盘的数量还没有达到 MAX_UPDATES_PER_SECOND。所以从优化的角度来讲，仍需要优化磁盘的读写性能。

第二种情况是 Relay 本身的性能问题，当 Relay 的发送队列数据超过 MAX_QUEUE_SIZE，它就会停止接收 metrics 或者丢弃数据。对此，可采用如下解决方法：第一是增加 MAX_QUEUE_SIZE 值，使其能够缓存更多的数据；第二是增加 Relay 后端的 Cache 或者 Relay 数量，这样前段的 Relay 能够将数据发送到更多的后端，避免数据堆积过多。

Carbon Cache、Carbon Relay 与 Aggregator 在转发数据的同时，也会将自身的一些性能指标发送出去。通过监控这些 metrics，就能实时地获得当前 Graphite Cluster 的性能和工作情况。这些数据在 metrics 层级树下的 Carbon 目录中。

relays 目录下是每个 Relay 实例的数据，其中比较重要的 metrics 有：

- ❑ metricsReceived：代表这个 Relay 实例收到的 metrics 数目。

- ❑ cpuUsage/memUsage：此 Relay 的 CPU 和内存使用状况。

- ❑ destinations.DESTINATION_NAME.relayMaxQueueLength/fullQueueDrops：这里的 DESTINATION_NAME 是 Relay 转发的目的地。这两个 metrics 可以看出 Relay 是否由于发送队列满而丢弃 metrics，一般情况下 fullQueueDrops 数值为零或者没有值，当有 metrics 有丢弃数据的时候，fullQueueDrops 会显示丢弃了多少数据，此时 relayMaxQueueLength 对应数值为 carbon.conf 中配置的 MAX_QUEUE_SIZE。

agents 目录下是每个 Carbon Cache 实例的数据，比较重要的有：

- ❑ updateOperations/pointsPerUpdate/avgUpdateTime：这几个 metrics 能够看出 Carbon Cache 写到 Whisper 数据库中的效率。updateOperations 是每分钟写入 Whisper 文件

的次数；avgUpdateTime 表示平均写入时间；显然，updateOperations 越多，表示写入性能越好；avgUpdateTime 越小，写入速度越快。pointsPerUpdate 表示每次将一个 metrics 写入 Whisper 文件时，写入 metrics 数据点的数目，这个数值最能反映当前磁盘接收大量数据写入的性能，其值应该是越小越好，因为值越小，表示在内存中保存的数据点越少，即数据会更快地写入磁盘。根据这个数值，还可简单地估算内存中保存了最近多长时间的 metrics。比如我们的 metrics 1 分钟发送一次，而此时 pointsPerUpdate 为 60，那么意味着内存中保存了最近 1 个小时的数据没有写到磁盘，如果 Carbon Cache 此时由于未知原因崩溃，那么大约会丢失最近一个小时的数据。

❑ metricsReceived：此 Carbon Cache 实例接收到的 metrics 数目。

❑ cpuUsage/memUsage：CPU 和内存的使用情况。

强烈建议用户在自己的生产环境中为 Graphite 自己的性能指标建立一个 Dashboard，以便对 Graphite 的性能进行方便的调试和监控。

7.8 其他

7.8.1 Whisper 文件操作

我们安装的 Whisper 数据库中自带了一些可以用来操作 Whisper 文件的 Python 脚本。它们对了解和调试 Graphite 的数据源有重要的作用。

❑ whisper-info.py

这个脚本用来查看 Whisper 数据文件的基本信息。比如文件大小、数据保存期等。示例如下：

```
[root@graphite01 LoadAverage]# whisper-info.py 15min.wsp |head -10
maxRetention: 31536000
xFilesFactor: 0.5
aggregationMethod: average
fileSize: 645172
```

```
Archive 0
retention: 604800
secondsPerPoint: 60
points: 10080
size: 120960
```

❑ whisper-dump.py

这个脚本用来显示所有保存在 Whisper 文件中的数据点。比如：

```
[root@graphite01 LoadAverage]# whisper-dump.py 5min.wsp |less
... ..
Archive 2 info:
```

```
offset: 224692
seconds per point: 900
points: 35040
retention: 31536000
size: 420480
```

Archive 0 data:

```
0: 1423145820,      11
1: 1423145880,      0
2: 1423145940,      8
3: 1423146000,      1
4: 1423146060,      2
5: 1423146120,      5
```

❑ whisper-create.py

这个脚本用来创建一个 Whisper 数据文件。比如：

```
[root@graphite01 LoadAverage]# whisper-create.py ./t.wsp 15m:8
Created: ./t.wsp (124 bytes)
```

❑ whisper-merge.py

这个脚本用来合并两个 Whisper 文件。

❑ whisper-resize.py

这个脚本用来调整 Whisper 文件的精度和保存期。比如一个 metrics 原来设置为每 5 分钟发送一次，但是后来发现需要改成每分钟发送一次，此时需要做两件事情。首先是更改 storage-schema.conf 中对应的配置，其次是需要调用这个脚本，对 Whisper 文件做一次修改，Graphite 没有办法自动调整 Whisper 文件参数。这个命令的使用方法为：whisper-resize.py filename timePerPoint:timeToStore，比如：

```
[root@graphite01 LoadAverage]# whisper-resize.py 5min.wsp 1m:1d
Retrieving all data from the archives
Creating new whisper database: 5min.wsp.tmp
Created: 5min.wsp.tmp (17308 bytes)
Migrating data without aggregation...
Renaming old database to: 5min.wsp.bak
Renaming new database to: 5min.wsp
```

7.8.2 压力测试

一般来说如果构建了一个 Graphite Cluster，会建议对其进行一个简单的压力测试，用以获得系统可以承受的最大压力，找出系统的瓶颈，再基于当前的硬件资源数据，获得系统以后扩容所需要的硬件数量。

一个简单的测试 Graphite 压力的方法是对 Graphite 系统发送大量的 metrics。可以使用以下脚本：

```
#!/usr/bin/env python",
```



```

import sys
import time
metrics = [
    "Disk.sda.BytesWritten 4083914240 ", "Memory.Cached 1269728 ", "Disk.sda2.Writes 14076569 ",
    "Memory.FreeSwap 4183572 ", "Disk.sda1.BytesWritten 270848 ", "Disk.sda1.Reads 190 ",
    "Memory.Swap 4192956 ", "Disk.sda5.Writes 57256269 ", "Cpu.System 831556819 ",
    "Cpu.Nice 777 ", "Disk.sda3.Writes 73216247 ", "Network.eth0.UnicastPktsIn 4163959471 ",
    "Cpu.Idle 3829019252 ", "Cpu.SoftInterrupts 20313884 ", "Disk.sda5.BytesWritten 1512787968 ",
    "Disk.sda1.Writes 51 ", "Cpu.Idle_Percent 87 ", "Cpu.User_Percent 11 ",
    "Disk.sda.Writes 144549136 ", "Cpu.User 2337308794 ", "Network.eth0.NonUnicastPktsIn 30 ",
    "Disk.sda3.BytesWritten 569655296 ", "Disk.sda3.Reads 121411674 ", "LoadAverage.5min 4.75 ",
    "Disk.sda4.BytesRead 4096 ", "Memory.Buffered 302588 ", "Disk.sda.Reads 122941876 ",
    "Disk.sda5.Reads 1504640 ", "Cpu.Kernel 809181260 ", "Cpu.HWInterrupts 2061675 ",
    "Disk.sda2.Reads 25346 ", "LoadAverage.1min 3.88 ", "Cpu.IOWait_Percent 2 ",
    "Disk.sda4.Reads 4 ", "Cpu.IOWait 151701631 ", "LoadAverage.15min 5.18 ",
    "Network.eth0.UnicastPktsOut 3986710449 ", "Cpu.SwapOut 78782276 ", "Memory.Ram 24626356 ",
    "Disk.sda2.BytesRead 174424576 ", "Disk.sda5.BytesRead 154915840 ", "Disk.sda3.BytesRead 1062800384 ",
    "Disk.sda2.BytesWritten 2001200128 ", "Disk.sda.BytesRead 1394321408 ", "Cpu.IOREceived 2790474728 ",
    "Disk.sda1.BytesRead 1955328 ", "Network.eth0.OctetsIn 1886501829 ", "Memory.FreeRam 26861932 ",
    "Network.eth0.OctetsOut 3030423715 ", "Cpu.ContextSwitches 1515325785 "]

metrics = ["testing.servers.server%d.%s %s" % i for i in metrics]

if len(sys.argv) != 2:
    print "Usage: %s metrics_number_to_generate" % sys.argv[0]
    exit(0)

for i in range(0, int(sys.argv[1])/len(metrics)):
    t = int(time.time())
    for j in metrics:
        print j % (i, t)

```

比如要产生 200 万个 metrics，将其发送给 Graphite，使用方法为：

```
python metrics.py 2000000 | nc graphite_host graphite_port
```

也可以将这个脚本设置成为 cron，定时重复地给 Graphite 发送数据。根据之前章节所述，可以关注诸如 pointsPerUpdate、metricsReceived 等 Graphite 本身的数据指标，此外还

需要关注 Graphite 物理服务器本身的 I/O、CPU、内存等性能数据，找出系统的瓶颈。

7.8.3 其他工具

得益于 Graphite 的简单易用，Graphite 的生态系统非常繁荣，除了之前提到的 Grafana 之外，还有许多工具可以和 Graphite 结合到一起，提供更强大的功能，如下。

- ❑ collectd：一个安装在客户机上，通过一个额外的 plugin 向 Graphite 发送 metrics 的工具。
- ❑ statsd：基于 Node.js 的数据聚合工具。
- ❑ graphite-to-zabbix：将 Graphite 数据发送给 Zabbix 的工具。
- ❑ graphite-beacon：一个检查 Graphite 数据并产生告警的工具。

在 <https://graphite.readthedocs.org/en/latest/tools.html> 可以查找到更多和 Graphite 结合在一起的工具，请读者自行查阅。

系统大规模部署

8.1 概述

系统就像细胞，是组成庞大功能集群的一个基础单元。系统的安装与调试便是系统管理员日常工作之一。从以前的光盘引导和网络启动，到现在的 Docker 集群应用，系统部署也在朝着自动化飞速奔跑。

8.2 与 PXE 不得不说的故事

8.2.1 PXE 简介

Preboot eXecution Environment (PXE)，是由 Intel 公司开发的，基于 C/S 网络模式下的网络系统安装技术。正是 PXE 的出现让系统管理员彻底摆脱了使用光驱、软驱、USB 移动设备进行操作系统安装的方式，同时提高了批量服务器安装的速度，简化了系统管理员的工作步骤。

下面介绍 PXE 的启动流程。

首先是 Client BIOS 启动，加载带有 PXE 支持的网卡。

然后 Client 网卡发送 DHCP 请求，请求 IP 地址信息及 Bootstrap 信息。Bootstrap protocol 详解请参看 http://en.wikipedia.org/wiki/Bootstrap_Protocol。这时，DHCP 服务器会返回 Client 的 IP 以及 Bootstrap 文件存放地址。

之后 Client 通过 TFTP 服务获取 Bootstrap 文件，并在本地加载 Bootstrap 文件。

在通过 TFTP 服务加载 Bootstrap 文件中制定的内核和文件系统后，就开始系统安装了。

8.2.2 PXE 实战

实战环境：CentOS 6.2 64bit

准备工作：VirtualBox、已安装好 CentOS 6 操作系统的虚拟机一台。

1. 配置测试客户端

打开 VirtualBox，单击创建按钮来创建一个新的 Client 测试虚拟机，如图 8-1 所示。

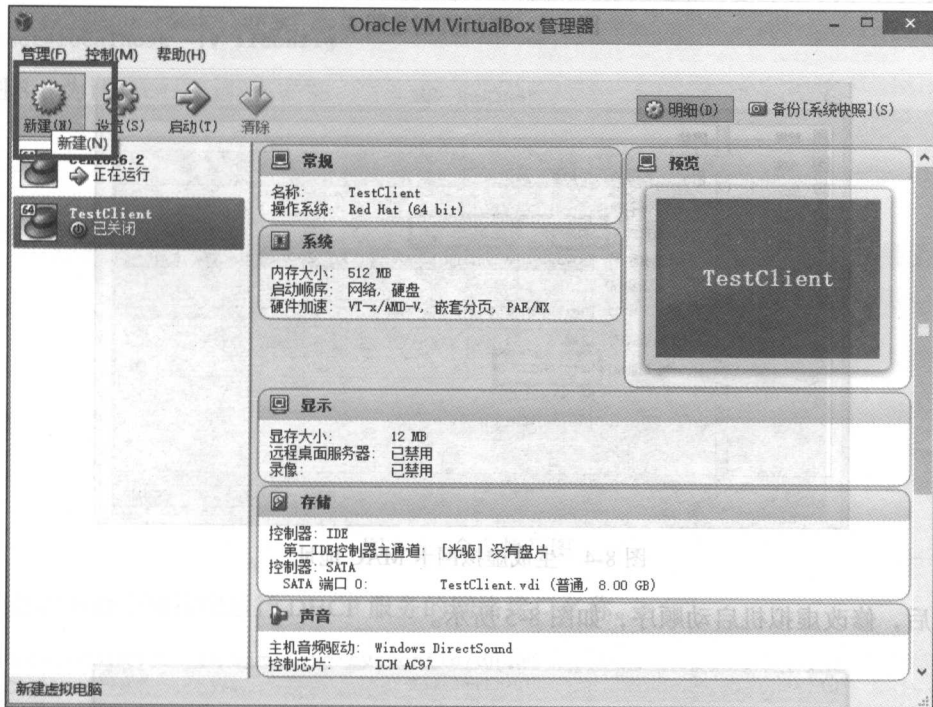


图 8-1 测试机创建图 1

然后在图 8-2 所示的界面输入虚拟电脑名称，配置类型为 Linux，版本为 Red Hat (64bit)。

在图 8-3 所示的界面配置 TestClient 的内存大小、硬盘大小、硬盘文件存放位置。此处可使用 VirtualBox 提供的默认值。最后点击创建，一个虚拟机就创建完成了。

在修改 TestClient 的配置时，可点击图 8-3 所示的按钮。

然后修改虚拟网卡连接方式为：桥接网卡，并生成虚拟网卡 MAC 地址。该地址会在后面服务端配置时使用（如图 8-4 所示）。

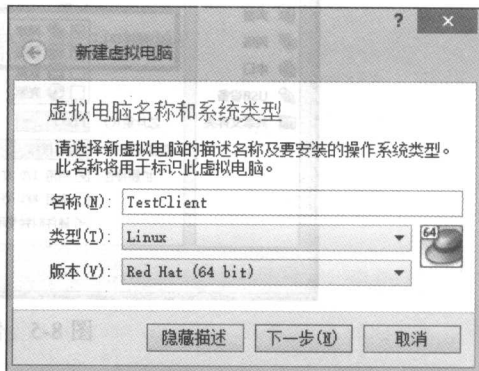


图 8-2 测试机创建图 2

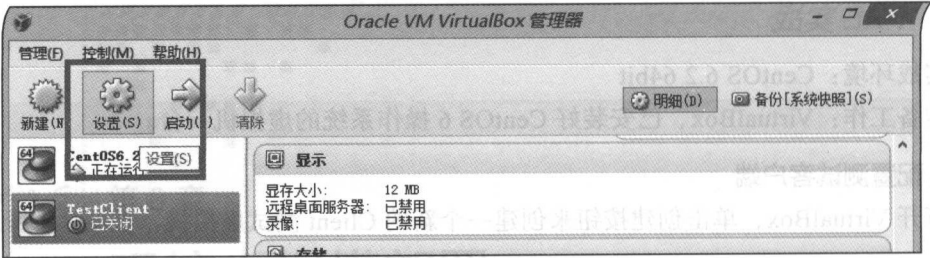


图 8-3 测试机创建图 3

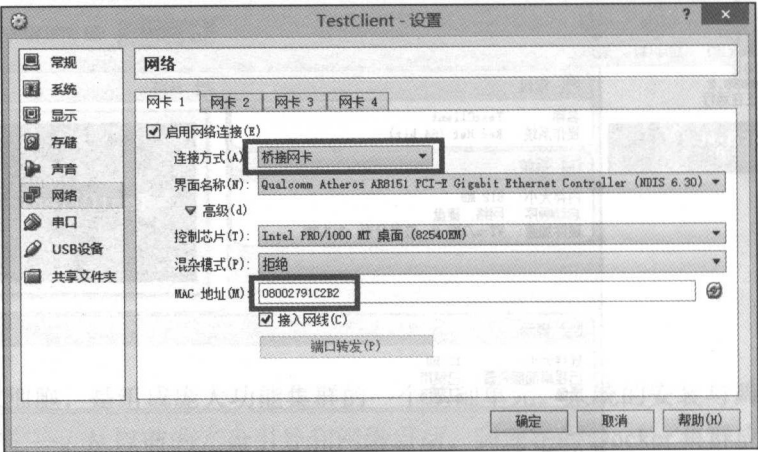


图 8-4 生成虚拟网卡 MAC 地址

最后，修改虚拟机启动顺序，如图 8-5 所示。

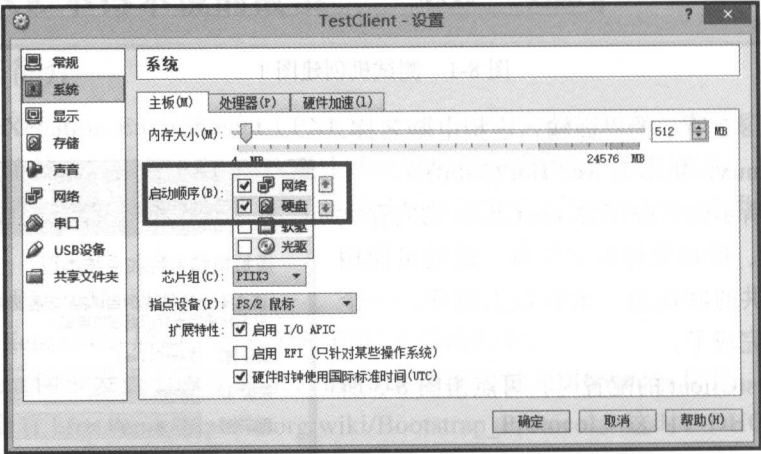


图 8-5 修改虚拟机启动顺序

至此 TestClient 配置完成。

2. 配置服务端 DHCP 服务

Client 为从网络启动的全新硬件，没有任何操作系统支持。所以在网络启动的时候需要配置相应的 DHCP 服务来接管 Client 的启动流程以及后续步骤。

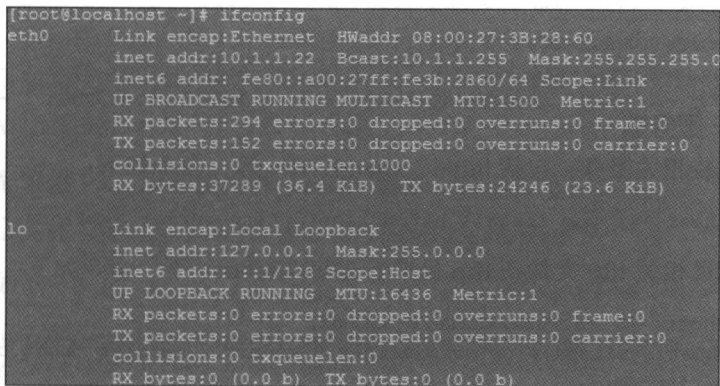
可通过以下命令安装 DHCP 服务包。

```
[root@localhost ~]# yum install dhcp -y
```

查看网络状态的命令如下：

```
[root@localhost ~]# ifconfig
```

图 8-6 为该命令执行状态截图。



```
[root@localhost ~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:3B:28:60
          inet addr:10.1.1.22  Bcast:10.1.1.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe3b:2860/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:294 errors:0 dropped:0 overruns:0 frame:0
          TX packets:152 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:37289 (36.4 KiB)  TX bytes:24246 (23.6 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

图 8-6 命令执行图

通过以下命令可指定运行 DHCP 服务的网络接口名称。

```
[root@localhost ~]# vim /etc/sysconfig/dhcpd
# Command line options here
DHCPDARGS=eth0
```

以下命令用来配置 DHCP 配置文件。

```
[root@localhost ~]# vim /etc/dhcp/dhcpd.conf
# Use this to enable / disable dynamic dns updates globally.
allow booting;
allow bootp;
option option-128 code 128 = string;
option option-129 code 129 = text;
filename "pxelinux.0";
next-server 10.1.1.22;
ddns-update-style none;
option domain-name "test.com";
option domain-name-servers ns1.test.com;
default-lease-time 600;
max-lease-time 7200;
subnet 10.1.1.0 netmask 255.255.255.0 {
```

```

range 10.1.1.100 10.1.1.150;
option domain-name-servers ns1.test.com;
option domain-name "test.com";
option routers 10.1.1.1;
option broadcast-address 10.1.1.255;
default-lease-time 600;
max-lease-time 7200;
filename "pxelinux.0";
next-server 10.1.1.22;
}
host testclient {
    hardware ethernet 08:00:27:91:C2:B2;
    fixed-address 10.1.1.105;
    filename "pxelinux.0";
    next-server 10.1.1.22;
}

```

3. 配置 PXE 服务端

第一步，安装相关服务，包括 httpd、xinetd、tftp 等，命令如下：

```
[root@localhost ~]# yum install httpd xinetd tftp-server syslinux -y
```

第二步，拷贝相关 tftp 文件到 tftp 目录 /var/lib/tftpboot/ 下，命令如下：

```

[root@localhost ~]# cd /usr/share/syslinux/
[root@localhost syslinux]# cp -r pxelinux.0 menu.c32 memdisk mboot.c32 chain.c32 /
var/lib/tftpboot/

```

第三步，修改 Xinetd 服务，启用 tftp。将配置文件中 disable 的值修改为 no，命令如下：

```

[root@localhost syslinux]# vim /etc/xinetd.d/tftp
service tftp
{
    socket_type          = dgram
    protocol             = udp
    wait                = yes
    user                 = root
    server               = /usr/sbin/in.tftpd
    server_args          = -s /var/lib/tftpboot
    disable              = no
    per_source           = 11
    cps                  = 100 2
    flags                = IPv4
}

```

第四步，创建用于启动的镜像目录，假定镜像存放位置为 /mnt/ 目录下。

```

[root@localhost ~]# cd /var/lib/tftpboot/
[root@localhost tftpboot]# mkdir centos6
[root@localhost mnt]# mount -o loop /mnt/CentOS-6.2-x86_64-bin-DVD1.iso /var/lib/

```

```
tftpboot/centos6
[root@localhost mnt]# ll /var/lib/tftpboot/centos6/
total 658
-rw-r--r-- 2 root root      14 Dec 16 2011 CentOS_BuildTag
drwxr-xr-x 3 root root    2048 Dec 11 2011 EFI
-rw-r--r-- 2 root root     212 Dec 15 2011 EULA
-rw-r--r-- 2 root root   18009 Dec 15 2011 GPL
drwxr-xr-x 3 root root    2048 Dec 11 2011 images
drwxr-xr-x 2 root root    2048 Dec 11 2011 isolinux
drwxrwxr-x 2 500 500 630784 Dec 16 2011 Packages
-rw-r--r-- 2 root root   1354 Dec  9 2011 RELEASE-NOTES-en-US.html
drwxr-xr-x 2 root root    4096 Dec 16 2011 repodata
-rw-r--r-- 2 root root   1706 Dec  9 2011 RPM-GPG-KEY-CentOS-6
-rw-r--r-- 2 root root   1730 Dec  9 2011 RPM-GPG-KEY-CentOS-Debug-6
-rw-r--r-- 2 root root   1730 Dec  9 2011 RPM-GPG-KEY-CentOS-Security-6
-rw-r--r-- 2 root root   1734 Dec  9 2011 RPM-GPG-KEY-CentOS-Testing-6
-r--r--r-- 1 root root   3380 Dec 16 2011 TRANS.TBL
[root@localhost mnt]#
```

第五步，配置供 PXE 服务使用的 Apache 服务，命令如下：

```
[root@localhost ~]# vim /etc/httpd/conf.d/pxeboot.conf
```

文件内容如下：

```
Alias /centos6 /var/lib/tftpboot/centos6
<Directory /var/lib/tftpboot/centos6>
Options Indexes FollowSymLinks
Allow from all
</Directory>
```

第六步，创建存放 PXE 配置文件的目录，并且创建默认的配置文件。

```
[root@localhost ~]# mkdir /var/lib/tftpboot/pxelinux.cfg
[root@localhost ~]# vim /var/lib/tftpboot/pxelinux.cfg/default
default menu.c32
prompt 0
timeout 60
menu title ### PXE Booting ###
label 1
menu label Install CentOS 6 64 bit system
kernel centos6/images/pxeboot/vmlinuz
append initrd=centos6/images/pxeboot/initrd.img method=http://10.1.1.22/centos6
devfs=nomount
```

最后，重启 PXE 相关服务，命令如下：

```
[root@localhost ~]# /etc/init.d/dhcpd restart
[root@localhost ~]# /etc/init.d/xinetd restart
[root@localhost ~]# /etc/init.d/httpd restart
```

现在，可以启动 TestClient 进行测试了。TestClient 可以通过 DHCP 获得相应 IP 地址，

并且通过 PXE 服务提供的网络安装进行系统安装。

是否已经对上述繁琐的配置感到厌烦？特别是当被管理的服务器数目达到一定的数量时，服务器信息修改的时间成本将大幅增加。下一节介绍的 Cobbler 将解决上述问题。

8.3 系统部署工具 Cobbler

8.3.1 Cobbler 简介

Cobbler 是由 Python 语言开发的系统部署工具。它提供了对 PXE 网络启动、DHCP、DNS、TFTP 的自动化管理，它支持对不同操作系统的快速安装，支持批量的 kickstart 分类管理，连接不同种类的服务器电源管理，yum 仓库管理、简单的 CMS 结构，并且也支持 KVM 上虚拟机的快速安装。系统安装过程中只要想得到的，基本都可以在 Cobbler 中找到相关功能。

Cobbler 的基础结构如图 8-7 所示^①。

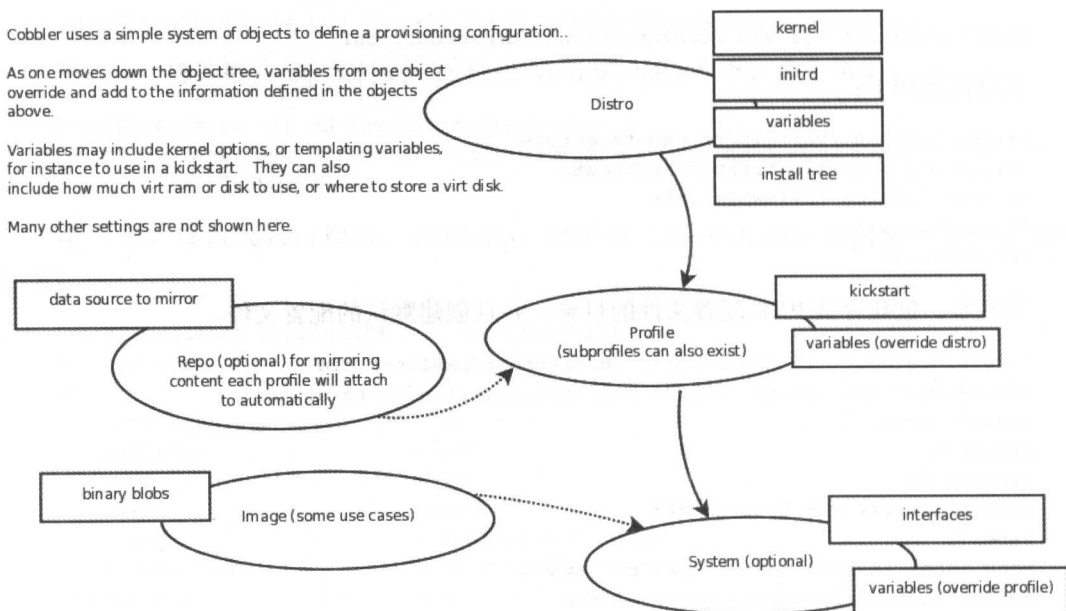


图 8-7 Cobbler 的基础结构

8.3.2 Cobbler 安装

1. EPEL 安装

CentOS 6 的官方 yum 仓库里不包含 Cobbler 的安装包，需要自行安装 Fedora EPEL

① 摘自 Cobbler 官网：https://cobbler.github.io/manuals/2.6.0/1_-_About_Cobbler.html

支持。

首先找到最新的 `epel-release` 包，下载并安装到 CentOS 上，下载地址如下：

`http://download.fedoraproject.org/pub/epel/6/x86_64/epel-release-X-X.noarch.rpm`

请注意使用最新的 `epel` 版本进行替换，本书以 `epel-release-6-8.noarch.rpm` 作为示例，代码如下：

```
[root@localhost ~]# wget http://download.fedoraproject.org/pub/epel/6/x86_64/
epel-release-6-8.noarch.rpm
[root@localhost ~]# rpm -ivh epel-release-6-8.noarch.rpm
warning: epel-release-6-8.noarch.rpm: Header V3 RSA/SHA256 Signature, key ID
0608b895: NOKEY
Preparing... ##### [100%]
1:epel-release ##### [100%]
[root@localhost ~]# yum update
Loaded plugins: fastestmirror
Setting up Update Process
Loading mirror speeds from cached hostfile
epel/metalink | 2.8 kB 00:00
* base: mirrors.aliyun.com
* epel: mirrors.neusoft.edu.cn
* extras: mirrors.163.com
* updates: mirrors.163.com
epel | 4.4 kB 00:00
epel/primary_db | 6.6 MB 01:12
No Packages marked for Update
[root@localhost ~]# yum
```

2. Cobbler 安装

在安装 EPEL 支持以后，Cobbler 就可以使用 `yum` 进行安装配置了，命令如下：

```
[root@localhost ~]# yum search cobbler
cobbler-web.noarch : Web interface for Cobbler
cobbler.noarch : Boot server configurator
koan.noarch : Helper tool that performs cobbler orders on remote machines
[root@localhost ~]# yum install -y cobbler cobbler-web
```

最新的 Cobbler 版本是存放在 `updates-testing` 仓库中的，所以我们需要对 Cobbler 进行版本升级，升级命令如下：

```
[root@localhost ~]# yum update --enablerepo=updates-testing cobbler cobbler-web
```

至此，Cobbler 已经安装完毕。

8.3.3 Cobbler 配置

1. Cobbler 的基础配置

启动 Cobbler 以后，服务是不可以直接使用的，下面开始进行 Cobbler 的配置工作。

首先启动 Cobbler 服务，命令如下：


```
[root@localhost ~]# /etc/init.d/cobblerd start
Starting cobbler daemon: [ OK ]
```

这时候的 Cobbler 是无法正常工作的，它提供了自带的 `check` 命令来进行运行监测，将已存在的问题标记出来，如下所示：

```
[root@localhost ~]# cobbler check
The following are potential configuration items that you may want to fix:

1 : The 'server' field in /etc/cobbler/settings must be set to something other than localhost, or kickstarting features will not work. This should be a resolvable hostname or IP for the boot server as reachable by all machines that will use it.
2 : For PXE to be functional, the 'next_server' field in /etc/cobbler/settings must be set to something other than 127.0.0.1, and should match the IP of the boot server on the PXE network.
3 : some network boot-loaders are missing from /var/lib/cobbler/loaders, you may run 'cobbler get-loaders' to download them, or, if you only want to handle x86/x86_64 netbooting, you may ensure that you have installed a *recent* version of the syslinux package installed and can ignore this message entirely. Files in this directory, should you want to support all architectures, should include pxelinux.0, menu.c32, elilo.efi, and yaboot. The 'cobbler get-loaders' command is the easiest way to resolve these requirements.
4 : change 'disable' to 'no' in /etc/xinetd.d/rsync
5 : debmirror package is not installed, it will be required to manage debian deployments and repositories
6 : ksvalidator was not found, install pykickstart
7 : The default password used by the sample templates for newly installed machines (default_password_crypted in /etc/cobbler/settings) is still set to 'cobbler' and should be changed, try: "openssl passwd -1 -salt 'random-phrase-here' 'your-password-here'" to generate new one
8 : fencing tools were not found, and are required to use the (optional) power management features. install cman or fence-agents to use them
```

Restart cobblerd and then run 'cobbler sync' to apply changes.

按照上面的提示，进行 Cobbler 配置的修复。

1) 修改配置文件中“server”项目的配置，将默认的 127.0.0.1 替换为本机的 IP 地址。

```
[root@localhost ~]# vim /etc/cobbler/settings
.....
# this is the address of the cobbler server -- as it is used
# by systems during the install process, it must be the address
# or hostname of the system as those systems can see the server.
# if you have a server that appears differently to different subnets
# (dual homed, etc), you need to read the --server-override section
# of the manpage for how that works.
# server: 127.0.0.1
server: 10.1.1.55
.....
```

2) 修改配置文件中“next_server”项目配置, 将默认的 127.0.0.1 替换为本机 IP 地址。

```
# if using cobbler with manage_dhcp, put the IP address
# of the cobbler server here so that PXE booting guests can find it
# if you do not set this correctly, this will be manifested in TFTP open timeouts.
next_server: 127.0.0.1
next_server: 10.1.1.55
```

3) 使用 Cobbler 提供的命令来初始化 boot-loader。这是 Cobbler 为了方便用户提供的
一个特性, 此特性需要将 Cobbler 版本升级到最新的稳定版, 否则不能使用。

```
[root@localhost ~]# cobbler get-loaders
task started: 2015-06-28_150427_get_loaders
task started (id=Download Bootloader Content, time=Sun Jun 28 15:04:27 2015)
path /var/lib/cobbler/loaders/README already exists, not overwriting existing
content, use --force if you wish to update
path /var/lib/cobbler/loaders/COPYING.elilo already exists, not overwriting
existing content, use --force if you wish to update
```

4) 在 xinetd 中启用 rsync 服务。

```
[root@localhost ~]# vim /etc/xinetd.d/rsync
.....
disable = no
.....

[root@localhost ~]# /etc/init.d/xinetd restart
Stopping xinetd:          [ OK ]
Starting xinetd:          [ OK ]
```

5) 安装 debmirror 包来管理 Deb 包类型。

```
[root@localhost ~]# yum install -y debmirror
[root@localhost ~]# vim /etc/debmirror.conf
```

然后修改 debmirror.conf 中的 dists 和 arches 配置。

```
.....
#@dists="sid";
#@arches="i386";
.....
```

安装 pykickstart 包的命令如下:

```
[root@localhost ~]# yum install -y pykickstart
```

6) 修改 Cobbler 默认安装系统的 admin 密码。

首先使用 openssl 生成加密后的密码, 下面以 cobblerpassword 为例进行说明。

```
[root@localhost ~]# openssl passwd -1 -salt 'random-phrase-here' 'cobblerpassword'
$1$random-p$XrhVlY7gRlJ/apYC2AMwq.
[root@localhost ~]# vim /etc/cobbler/settings
#default_password_crypted: "$1$mF86/UHC$WvcIcX2t6crBz2onWxyac."
```

```
default_password_crypted: "$1$random-p$XrhVlY7gRlJ/apYC2AMwq."
```

7) 安装 cman 来进行电源管理。

```
[root@localhost ~]# yum install -y cman
```

重启 Cobbler 服务，再次运行 Cobbler 的 check 命令来检查 Cobbler 服务的状态。

```
[root@localhost ~]# /etc/init.d/cobblerd restart
Stopping cobbler daemon:          [ OK ]
Starting cobbler daemon:          [ OK ]
[root@localhost ~]# cobbler check
No configuration problems found. All systems go.
```

2. Cobbler 的高级配置

1) 检查并禁用 SELinux 功能。

```
[root@localhost ~]# getsebool
getsebool: SELinux is disabled
```

2) 关闭 IPtables (对网络有专业知识的人可保留、添加相应规则)。

```
[root@localhost ~]# /etc/init.d/iptables stop
iptables: Setting chains to policy ACCEPT: filter      [ OK ]
iptables: Flushing firewall rules:                    [ OK ]
iptables: Unloading modules:                          [ OK ]
[root@localhost ~]# chkconfig iptables off
```

3) 配置 Cobbler 来管理 DHCP 功能。

先修改 /etc/cobbler/settings 配置文件，启用 DHCP 管理。

```
.....
manage_dhcp: 1
.....
```

然后修改 Cobbler 用来管理 DHCP 的模版文件 /etc/cobbler/dhcp.template。该模板主要是 Cobbler 用来生成 dhcpd.conf 文件的，所以需要修改相应的子网信息、网关信息、DNS 信息，以及 DHCP 地址池信息。如果要管理多个子网，只需再添加 subnet 字典部分即可，如下所示：

```
subnet 10.1.1.0 netmask 255.255.255.0 {
    option routers          10.1.1.1;
    option domain-name-servers 10.1.1.55;
    option subnet-mask      255.255.255.0;
    range dynamic-bootp     10.1.1.100 10.1.1.110;
    default-lease-time      21600;
    max-lease-time          43200;
    next-server              $next_server;
    class "pxeclients" {
        match if substring (option vendor-class-identifier, 0, 9) = "PXEClient";
```

```

if option pxe-system-type = 00:02 {
    filename "ia64/elilo.efi";
} else if option pxe-system-type = 00:06 {
    filename "grub/grub-x86.efi";
} else if option pxe-system-type = 00:07 {
    filename "grub/grub-x86_64.efi";
} else {
    filename "pxelinux.0";
}
}
}

```

4) 配置 Cobbler 来管理 tftp 功能。

首先修改 `/etc/cobbler/settings` 配置文件，启用 tftp 管理。

```

.....
manage_tftpd: 1
.....

```

然后调整 tftp 的配置文件模板。cps 参数可根据需要同时安装的服务器数目来进行适当的增加。cps 两个参数的含义：第一个参数为 1 秒内可以同时处理的 tftp 请求数目，如果超过该数目，tftp 服务会被临时暂停服务。第二个参数用于确定在 tftp 被临时暂停服务后多长时间可以重新启用。

```

[root@localhost ~]# vim /etc/cobbler/tftpd.template
service tftp
{
    disable                = no
    socket_type             = dgram
    protocol               = udp
    wait                   = yes
    user                   = $user
    server                 = $binary
    server_args             = -B 1380 -v -s $args
    per_source              = 11
    cps                    = 100 2
    flags                  = IPv4
}

```

5) 使用 Cobbler 可以管理 DNS，但是通常用户已经搭建自己的 DNS 服务器，所以在本章中 Cobbler 不启用 DNS 管理功能。

8.3.4 Cobbler 应用

经过一番努力，Cobbler 服务已经基本配置完成。但是在导入需要的系统镜像以及待安装客户端信息之前，Cobbler 还不能提供系统的自动化安装服务。本节将展示如何使 Cobbler 开始提供系统安装服务。

1. 导入待安装的系统镜像

这里以 CentOS 6 为例进行讲解，首先将镜像挂载到本地文件系统，命令如下：

```
[root@localhost ~]# mount -t iso9660 -o loop,ro /root/CentOS-6.2-x86_64-bin-DVD1.iso /mnt
```

然后使用 **cobbler** 命令将镜像导入 Cobbler 系统。

```
[root@localhost ~]# cobbler import --name=CentOS6.2 --arch=x86_64 --path=/mnt
task started: 2015-06-29_021350_import
task started (id=Media import, time=Mon Jun 29 02:13:50 2015)
Found a candidate signature: breed=redhat, version=rhel6
Found a matching signature: breed=redhat, version=rhel6
Adding distros from path /var/www/cobbler/ks_mirror/CentOS6.2-x86_64:
creating new distro: CentOS6.2-x86_64
trying symlink: /var/www/cobbler/ks_mirror/CentOS6.2-x86_64 -> /var/www/cobbler/links/CentOS6.2-x86_64
creating new profile: CentOS6.2-x86_64
associating repos
checking for rsync repo(s)
checking for rhn repo(s)
checking for yum repo(s)
starting descent into /var/www/cobbler/ks_mirror/CentOS6.2-x86_64 for CentOS6.2-x86_64
processing repo at : /var/www/cobbler/ks_mirror/CentOS6.2-x86_64
need to process repo/comps: /var/www/cobbler/ks_mirror/CentOS6.2-x86_64
looking for /var/www/cobbler/ks_mirror/CentOS6.2-x86_64/repodata/*comps*.xml
Keeping repodata as-is :/var/www/cobbler/ks_mirror/CentOS6.2-x86_64/repodata
*** TASK COMPLETE ***
```

导入完成后，可以查看 Cobbler 状态，发现 Cobbler 自动添加了 distro 和 profile 项目。

```
[root@localhost ~]# cobbler list
distros:
    CentOS6.2-x86_64
profiles:
    CentOS6.2-x86_64
.....
```

2. 创建本地仓库（可选）

在具有特殊安全策略的机房中，不是所有的服务器都有公网访问权限。这种情况下需要在本地创建软件仓库。前提条件是，Cobbler 服务器需要有公网访问来进行软件仓库的下载。

可运行如下命令来添加软件仓库：

```
[root@localhost ~]# cobbler repo add --name=CentOS-updates --mirror=http://mirror.centos.org/centos-6/6/updates/x86_64/
```

然后运行如下命令来进行同步远程仓库：

```
[root@localhost ~]# cobbler reposync
```

```

task started: 2015-07-19_220832_reposync
task started (id=Reposync, time=Sun Jul 19 22:08:32 2015)
hello, reposync
run, reposync, run!
creating: /var/www/cobbler/repo_mirror/CentOS-updates/config.repo
creating: /var/www/cobbler/repo_mirror/CentOS-updates/.origin/CentOS-updates.
repo
running: /usr/bin/reposync -l -n -d --config=/var/www/cobbler/repo_mirror/
CentOS-updates/.origin/CentOS-updates.repo --repoid=CentOS-updates
--download_path=/var/www/cobbler/repo_mirror -a x86_64
.....
Saving Primary metadata
Saving file lists metadata
Saving other metadata
Saving delta metadata
Generating sqlite DBs
Sqlite DBs complete

received on stderr:
running: chown -R root:apache /var/www/cobbler/repo_mirror/CentOS-updates
received on stdout:
received on stderr:
running: chmod -R 755 /var/www/cobbler/repo_mirror/CentOS-updates
received on stdout:
received on stderr:
*** TASK COMPLETE ***

```

完成后，更新的软件包将会存放在 `/var/www/cobbler/repo_mirror/<REPO-NAME>` 下。

在 Cobbler 的配置文件中，修改下面的参数，可使系统安装完成后自动使用 Cobbler 作为软件更新仓库。

```
yum_post_install_mirror: 1
```

3. 创建系统

创建系统之前，需要准备如下内容。

- ☐ 安装服务器的网卡名称和 Mac 地址。
- ☐ 安装服务器的 IP 地址。
- ☐ 安装服务器的操作系统信息。

准备完成后即可在 Cobbler 中创建系统了，命令如下：

```
[root@localhost ~]# cobbler system add --name=test-server --profile=CentOS6.2-
x86_64 --interface=eth0 --mac=08:00:27:A6:5D:A5 --ip-address=10.1.1.56
--netmask=255.255.255.0 --gateway=10.1.1.1
```

通过如下命令查看已经创建的系统信息：

```
[root@localhost ~]# cobbler system report --name test-server
Name: test-server
```

```
TFTP Boot Files: {}
Comment:
Enable gPXE?: <<inherit>>
Fetchable Files: {}
Gateway: 10.1.1.1
.....
```

将客户端服务器 `test-server` 的网络连接好，开机。熟悉的安装界面就映入眼帘了。

如果在创建的时候忘记了某些参数的设置也没有关系，可使用相应的命令来修改。比如可以使用下面的命令来关闭系统的网络安装。

```
[root@localhost ~]# cobbler system edit --name test-server --netboot=n
```

变更之后，系统的 `Netboot Enabled` 参数变为 `False`，这样即使系统服务器重启，也不会再次进行系统安装。

```
[root@localhost ~]# cobbler system report --name test-server
.....
Netboot Enabled           : False
.....
```

4. 小结

(1) Distro Profile System 在 Cobbler 中的关系

完成 Cobbler 的安装配置以后，相信大家对 Cobbler 已经有了一定的了解，但这里还是要介绍 Cobbler 的一些结构和概念。

可以将 `distro` 比作一个学校，里面定义了如下参数：

```
[root@localhost ~]# cobbler distro report --name CentOS6_Test
Name: CentOS6_Test
Architecture: x86_64
TFTP Boot Files: {}
Breed: redhat
Comment:
Fetchable Files: {}
Initrd: /var/www/cobbler/ks_mirror/CentOS6_Test/initrd
Kernel: /var/www/cobbler/ks_mirror/CentOS6_Test/vmlinuz
Kernel Options: {}
Kernel Options (Post Install): {}
Kickstart Metadata: {}
Management Classes: []
OS Version: rhel6
Owners: ['admin']
Red Hat Management Key: <<inherit>>
Red Hat Management Server: <<inherit>>
Template Files: {}
```

`profile` 则可以比作学校中的班级，里面包含下面参数：

```
[root@stnd0001 ~]# cobbler profile report --name CentOS6_Diskless_development
```

```

Name: CentOS6_Test
TFTP Boot Files: {}
Comment:
DHCP Tag: default
Distribution: CentOS6_Test
Enable gPXE?: 0
Enable PXE Menu?: 0
Fetchable Files: {}
Kernel Options: {}
Kernel Options (Post Install): {}
Kickstart:
Kickstart Metadata: {}
Management Classes: []
Management Parameters: <<inherit>>
Name Servers: []
Name Servers Search Path: []
Owners: ['admin']
Parent Profile:
Proxy:
Red Hat Management Key: <<inherit>>
Red Hat Management Server: <<inherit>>
Repos: []
Server Override: <<inherit>>
Template Files: {}
Virt Auto Boot: 1
Virt Bridge: virbr0
Virt CPUs: 1
Virt Disk Driver Type: raw
Virt File Size(GB): 5
Virt Path:
Virt RAM (MB): 512
Virt Type: kvm

```

system 就是班级中的学生，包含的参数如下：

```

[root@stdn0001 ~]# cobbler system report --name stdn0001
Name: stdn0001
TFTP Boot Files: {}
Comment:
Enable gPXE?: 0
Fetchable Files: {}
Gateway: 10.1.1.1
Hostname: test-client
Image:
..... (省略部分)
Interface Type:
IP Address: 10.1.1.56
IPv6 Address:
IPv6 Default Gateway:
IPv6 MTU:
IPv6 Prefix:

```



```
IPv6 Secondaries: []
IPv6 Static Routes: []
MAC Address: 08:00:27:A6:5D:A5
Management Interface: False
MTU:
Subnet Mask: 255.255.255.0
Static: False
Static Routes: []
Virt Bridge:
```

所有的 system、profile 和 distro 都是继承关系。所有在 distro 中的参数都会被 profile 继承，再被 system 继承。但是当 distro 中的参数值和 profile 中不同时，distro 中的值会被重写。同样，system 中的值也会重写 profile 和 distro 中的值。

(2) 重要的配置文件目录

下面再介绍一下 Cobbler 的配置文件目录结构和内容。

Cobbler 的配置文件主要存放在 /var/lib/cobbler/ 目录下（默认安装情况）。其中，每个文件夹中主要包括如下内容。

- ❑ config/: 主要存放生成后的 system、profile、distro、repo 等配置文件。
- ❑ kickstarts/: 就像名字一样，这个目录主要存放被 Cobbler 引用的 ks 相关的文件。
- ❑ snippets/: 存放相关 snippet 脚本的文件夹。
- ❑ triggers/: 存放自动化脚本的文件夹。

这里主要介绍 trigger 目录的作用。add、change、delete、install、sync 表示在 Cobbler 执行某个行动的时候来调用目录下的脚本。每个目录下包含了操作所对应的对象，比如常用的 distro、profile 和 system。在这个目录下，还有 post 和 pre 分别表示在动作之后还是之前运行目录下的脚本。install 和 sync 是 Cobbler 直接执行的行动，所以目录下直接是 post 和 pre 目录。

下面举个例子：如果在 /var/lib/cobbler/trigger/sync/post/ 目录下放一个 shell 脚本，echo 'date' > /tmp/cobbler_sync_time.txt。那么，在每次运行 cobbler sync 命令后，就会将命令运行的时间写到 /tmp/cobbler_sync_time.txt 文件中。

8.3.5 Cobbler API

Cobbler 推荐使用 xmlrpclib 调用 API。下面直接使用一个例子来简单讲述如何使用 Cobbler API 进行管理。

将如下脚本放在刚安装完成的 Cobbler 服务器上，修改好用户名和密码就可以运行测试了。

```
#!/usr/bin/python2.7
import xmlrpclib
USERNAME = 'admin'
PASSWORD = 'cobblerpassword'
```

```

server = xmlrpclib.Server("http://localhost/cobbler_api")
print "Get remote data from Cobbler Server."
print "Distros in Cobbler"
print server.get_distros()
print "Profile in Cobbler"
print server.get_profiles()
print "Systems in Cobbler"
print server.get_systems()
print "Search distro in Cobbler"
print server.find_distro({"name": "Cent*"})
print "Get the token for modify cobbler information"
token = server.login(USERNAME, PASSWORD)
print token
print "Change distro comment"
handle = server.get_distro_handle('CentOS6.2-x86_64', token)
server.modify_distro(handle, 'comment', 'Testing modification', token)
server.save_distro(handle, token)

```

运行完成后，再使用 `cobbler distro report --name CentOS6.2-x86_64` 来查看 comment 是否已经变更为 “Testing modification”。

如果你得到如下报错信息，则说明 Cobbler 的用户名和密码没有正确配置。

```

Traceback (most recent call last):
  File "./a.py", line 8, in <module>
    token = server.login(USERNAME, PASSWORD)
  File "/usr/lib64/python2.7/xmlrpclib.py", line 1224, in __call__
    return self.__send(self.__name, args)
  File "/usr/lib64/python2.7/xmlrpclib.py", line 1578, in __request
    verbose=self.__verbose
  File "/usr/lib64/python2.7/xmlrpclib.py", line 1264, in request
    return self.single_request(host, handler, request_body, verbose)
  File "/usr/lib64/python2.7/xmlrpclib.py", line 1297, in single_request
    return self.parse_response(response)
  File "/usr/lib64/python2.7/xmlrpclib.py", line 1473, in parse_response
    return u.close()
  File "/usr/lib64/python2.7/xmlrpclib.py", line 793, in close
    raise Fault(**self._stack[0])
xmlrpclib.Fault: <Fault 1: "[class 'cobbler.cexceptions.CX']:'login failed
(admin)'">

```

这时，请按照前面的步骤重设 admin 密码。

8.3.6 Cobbler Replication

在主从流行的今天，Cobbler 也参与到高可用的阵营中来了。Slave 不仅可以提供高可用支持，还可以在批量服务器部署的时候降低 Master 的网络压力，提高服务器部署速度。如果单台 Cobbler 在进行服务器部署时达到了 TFTP 上限，那么需要添加一个或多个 Slave 来进行网络带宽分流。

下面介绍如何配置一台 *Slave* 服务器。

首先，按照本章前面的方法安装一台全新的带有 Cobbler 服务的服务器。

然后，使用 Cobbler 自带的 `replicate` 命令在 *Slave* 运行拉取 *Master* 上的所有数据，即可完成配置。

Replication 在如今产品化的服务中创建就是这么简单。

下面简单介绍下 Cobbler 的 `replicate` 命令。

```
[root@localhost ~]# cobbler replicate -h
Usage: cobbler [options]
Options:
  -h, --help                show this help message and exit
  --master=MASTER          Cobbler server to replicate from.
  --distros=DISTRO_PATTERNS
                           patterns of distros to replicate
  --profiles=PROFILE_PATTERNS
                           patterns of profiles to replicate
  --systems=SYSTEM_PATTERNS
                           patterns of systems to replicate
  --repos=REPO_PATTERNS    patterns of repos to replicate
  --image=IMAGE_PATTERNS   patterns of images to replicate
  --mgmtclasses=MGMTCLASS_PATTERNS
                           patterns of mgmtclasses to replicate
  --packages=PACKAGE_PATTERNS
                           patterns of packages to replicate
  --files=FILE_PATTERNS    patterns of files to replicate
  --omit-data               do not rsync data
  --sync-all               sync all data
  --prune                   remove objects (of all types) not found on the master
  --use-ssl                 use ssl to access the Cobbler master server api
```

常用的几个参数主要有 `--master`、`--sync-all` 和 `--prune`。`--master` 用来指定 cobbler master 的 IP 地址。`--sync-all` 用来同步所有 master 上的信息，包括了 `distro`、`profile`、`system` 等。`--prune` 用在 *slave* 上删除 cobbler master 上已经被删除的配置信息。

综上，我们在 *Slave* 上运行的命令如下：

```
cobbler replicate --master=COBBLER_MASTER_IP --sync-all --prune
```

8.3.7 Cobbler 实战

下面通过实战方式来配置一个 $N+1$ 的 Cobbler 集群，用来支持大批量的服务器部署需求， N 可以根据实际情况来进行设置。其基本结构如图 8-8 所示。

搭建步骤如下：

1) 网络设备在跨 VLAN 进行 DHCP 广播的时候，需要开启 `dhcp-relay` 以便接收到

DHCP 请求。

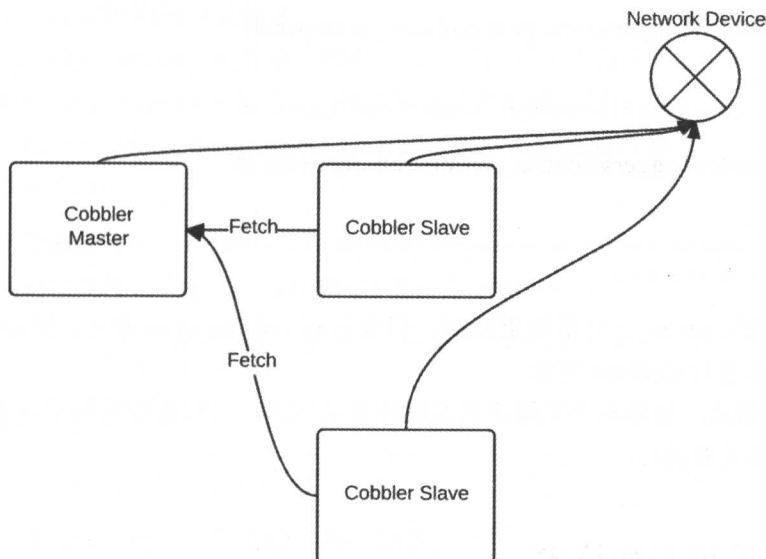


图 8-8 Cobbler 集群基本结构

- 2) 所有 Slave 服务器需要开启到 Master 服务器 HTTP 和 RSYNC 的网络策略。
- 3) 创建和配置 Master 服务器。包括镜像导入、系统导入等。
- 4) 创建和配置 Slave 服务器。测试 Replicate 命令是否成功。
- 5) 配置从 Master 服务器到所有 Slave 服务器的 root 免密码 ssh 登录。
- 6) 放置自动化脚本。

在 Master 上放置的自动化脚本路径和内容如下：

`/var/lib/cobbler/cobbler_scripts/replicate.sh`

```
#!/bin/bash
SLAVE_SERVERS=" slave01 slave02"
for i in $SLAVE_SERVERS
do
    ssh root@$i /var/lib/cobbler/cobbler_scripts/run_replicate.sh
done
```

`/var/lib/cobbler/triggers/sync/post/replicate_wrapper.sh`

```
#!/bin/bash
bash /var/lib/cobbler/cobbler_scripts/replicate.sh
```

在 Slave 上放置的自动化脚本路径和内容如下：

`/var/lib/cobbler/cobbler_scripts/replicate.sh`

```
#!/bin/bash
echo "Run replicate script" > /tmp/cobbler_replicate.log
```

```

echo `date` >> /tmp/cobbler_replicate.log

/var/lib/cobbler/triggers/sync/post/replicate_wrapper.sh

#!/bin/bash
bash /var/lib/cobbler/cobbler_scripts/replicate.sh

/var/lib/cobbler/triggers/cobbler_scripts/run_replicate.sh

#!/bin/bash
/usr/bin/cobbler replicate --master=MASTER_IP --sync-all --prune
/usr/bin/cobbler sync

```

如此每当在 Master 上有系统更新时，只要运行 `cobbler sync` 命令，Master 就会 ssh 到所有 Slave 上来进行 Cobbler 更新。

这是主动模式，如果对于系统更新及时性要求不高，可以在所有的 Cobbler Slave 上添加 `cronjob` 的模式来进行。

8.4 操作系统无盘技术

8.4.1 定义

本节中所提到的操作系统无盘技术实际使用的是自定义大小的内存作为操作系统根分区的挂载点，使用剩余部分作为系统内存，通过网络启动方式来完成系统镜像的传输和安装的一种技术。

优点：

- ❑ 节约成本。省去了购买硬盘的费用。
- ❑ 速度快。内存的读写速度是硬盘暂时无法比拟的。
- ❑ 安全性好。相对于使用硬盘，无盘系统在发生特殊情况下，可以降低数据外泄的概率。在已经断电的内存中恢复数据是非常难以实现的。
- ❑ 部署速度快。在完成无盘镜像的配置以后，批量部署速度较普通系统安装方式较快，而且可以保证配置的高度一致。
- ❑ 日常维护简单。在完成无盘镜像的配置以后，日常维护或者故障处理相对简单，只需重启服务器即可。

缺点：

非持久化存储。在使用内存作为操作系统存储介质时，服务器无法应用在有持久化存储需求的服务中，例如数据库（只读缓存除外）等。

8.4.2 制作无盘镜像

介绍完无盘系统的特点以后，下面来创建一个无盘操作系统。

1. 准备阶段

首先，定义需要使用的目录信息。

□ /root/diskless-image: 主目录，所有文件都将放在这个目录下。

□ /root/diskless-image/rootfs-mnt: 挂载镜像文件的临时目录。

□ /root/diskless-image/dracut-tmp: Dracut 工具使用的临时目录。

其次，准备 Dracut 工具。

2. 制作无盘镜像

创建无盘镜像文件，大小为 2GB=2048MB，命令如下：

```
[root@localhost diskless-image]# dd if=/dev/zero of=/root/diskless-image/rootfs.
img bs=1k count=$((2048 * 1024))
2097152+0 records in
2097152+0 records out
2147483648 bytes (2.1 GB) copied, 6.73247 s, 319 MB/s
[root@localhost diskless-image]#
```

格式化镜像文件并创建文件系统，命令如下：

```
[root@localhost ~]# cd /root/diskless-image/
[root@localhost diskless-image]# mke2fs -v -F -L ROOT rootfs.img
mke2fs 1.41.12 (17-May-2010)
fs_types for mke2fs.conf resolution: 'ext2', 'default'
Filesystem label=ROOT
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
131072 inodes, 524288 blocks
26214 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=536870912
16 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 29 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
[root@localhost diskless-image]#
```

以下是安装 Dracut 的步骤。

系统自带的 Dracut 版本较为落后，无法正常引导。所以需要去官网重新下载源码包进行编译安装。

官方文档以及需要的安装包参见：<https://www.kernel.org/pub/linux/utils/boot/dracut/dracut.html>

安装命令如下：

```
[root@localhost diskless-image]# wget https://www.kernel.org/pub/linux/utils/
boot/dracut/dracut-044.tar.gz
[root@localhost diskless-image]# tar -zxvf dracut-044.tar.gz
[root@localhost diskless-image]# cd dracut-044
[root@localhost dracut-044]# make -f Makefile
[root@localhost dracut-044]# make install
[root@localhost dracut-044]# cd
[root@localhost ~]#
```

可通过以下命令检查 Dracut 版本。

```
[root@localhost ~]# dracut -h
Usage: /usr/bin/dracut [OPTION]... [<initramfs> [<kernel-version>]]
Version: 044
```

下面使用 Dracut 创建启动时所需要的 initrd 文件。

```
[root@localhost diskless-image]# /usr/bin/dracut -a livenet -o 'i18n lvm dmraid
plymouth dropbear-sshd' -f initrd
dracut: Executing: /usr/bin/dracut -a livenet -o "i18n lvm dmraid plymouth
dropbear-sshd" -f initrd
dracut: dracut module 'bootchart' will not be installed, because command '/sbin/
bootchartd' could not be found!
dracut: dracut module 'systemd' will not be installed, because command '/systemd'
could not be found!
dracut: dracut module 'systemd-bootchart' will not be installed, because command
'/systemd-bootchart' could not be found!
dracut: systemd-initrd needs systemd in the initramfs
dracut: systemd-networkd needs systemd in the initramfs
dracut: dracut module 'i18n' will not be installed, because it's in the list to
be omitted!
dracut: dracut module 'plymouth' will not be installed, because it's in the list
to be omitted!
dracut: dracut module 'btrfs' will not be installed, because command 'btrfs'
could not be found!
dracut: dracut module 'dmraid' will not be installed, because it's in the list to
be omitted!
dracut: dracut module 'lvm' will not be installed, because it's in the list to be
omitted!
dracut: dracut module 'nbd' will not be installed, because command 'nbd-client'
could not be found!
dracut: dracut-systemd needs systemd-initrd in the initramfs
dracut: *** Including module: bash ***
dracut: *** Including module: dash ***
dracut: *** Including module: caps ***
dracut: *** Including module: modsign ***
dracut: *** Including module: network ***
```

```

dracut: *** Including module: ifcfg ***
dracut: *** Including module: url-lib ***
dracut: *** Including module: crypt ***
dracut: *** Including module: dm ***
dracut: Skipping udev rule: 60-persistent-storage-dm.rules
dracut: Skipping udev rule: 55-dm.rules
dracut: *** Including module: dmsquash-live ***
dracut: *** Including module: kernel-modules ***
dracut: *** Including module: kernel-network-modules ***
dracut: *** Including module: livenet ***
dracut: *** Including module: mdraid ***
dracut: Skipping udev rule: 63-md-raid-arrays.rules
dracut: Skipping udev rule: 64-md-raid-assembly.rules
dracut: *** Including module: multipath ***
dracut: Skipping program socket:/org/kernel/dm/multipath_event using in udev rule
40-multipath.rules as it cannot be found
dracut: *** Including module: cifs ***
dracut: *** Including module: fcoe ***
dracut: *** Including module: fcoe-uefi ***
dracut: *** Including module: iscsi ***
dracut: *** Including module: nfs ***
dracut: *** Including module: resume ***
dracut: *** Including module: rootfs-block ***
dracut: *** Including module: terminfo ***
dracut: *** Including module: udev-rules ***
dracut: Skipping udev rule: 91-permissions.rules
dracut: Skipping udev rule: 80-drivers-modprobe.rules
dracut: *** Including module: biosdevname ***
dracut: *** Including module: usrmount ***
dracut: *** Including module: base ***
/usr/lib/dracut/modules.d/99base/module-setup.sh: line 15: /var/tmp/dracut.
WGbXI0/initramfs/usr/lib/initrd-release: No such file or directory
ln: creating symbolic link `/var/tmp/dracut.WGbXI0/initramfs/usr/lib/os-release':
No such file or directory
dracut: *** Including module: fs-lib ***
dracut: *** Including module: img-lib ***
dracut: *** Including module: shutdown ***
dracut: *** Including module: uefi-lib ***
dracut: *** Including modules done ***
dracut: *** Installing kernel module dependencies and firmware ***
dracut: *** Installing kernel module dependencies and firmware done ***
dracut: *** Resolving executable dependencies ***
dracut: *** Resolving executable dependencies done***
dracut: *** Pre-linking files ***
dracut: *** Pre-linking files done ***
dracut: *** Stripping files ***
dracut: *** Stripping files done ***
dracut: *** Store current command line parameters ***
dracut: *** Creating image file '/root/diskless-image/initrd' ***
dracut: *** Creating initramfs image file '/root/diskless-image/initrd' done ***
[root@localhost diskless-image]#

```


然后将后面使用的 Kernel 也一起复制到该目录下。

```
[root@localhost diskless-image]# cp /boot/vmlinuz-2.6.32-573.22.1.el6.x86_64
vmlinuz
[root@localhost diskless-image]#
```

接着，创建镜像文件的挂载目录并挂载镜像文件。

```
[root@localhost diskless-image]# mkdir rootfs-mnt
[root@localhost diskless-image]# mount -o loop rootfs.img rootfs-mnt
[root@localhost diskless-image]# mkdir rootfs-mnt/{proc,dev}
[root@localhost diskless-image]# mount --bind /proc rootfs-mnt/proc
[root@localhost diskless-image]# mount --bind /dev rootfs-mnt/dev
[root@localhost diskless-image]#
```

下载并安装 centos-release 包。

```
[root@localhost diskless-image]# yum install -y --installroot=/root/diskless-
image/rootfs-mnt/ --releasever=6 centos-release
```

指定目录安装系统基本包以及相关基础包，命令如下：

```
[root@localhost rootfs-mnt]# yum --installroot=/root/diskless-image/rootfs-mnt/
--releasever=6 install findutils filesystem bash kernel passwd dhclient yum
openssh-server openssh-clients vim ntp rootfiles
```

下面使用 chroot 切换到无盘镜像的系统中。

```
[root@localhost diskless-image]# chroot /root/diskless-image/rootfs-mnt/
```

通过以下命令查看并检查 passwd 文件是否正常。

```
[root@localhost /]# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
[root@localhost /]#
```

以下是为 root 用户配置密码的步骤，密码与制作镜像的服务器 root 密码相同。

首先，切换到当前系统并获取 root 密码的加密字符串。

```
[root@localhost /]# exit
exit
[root@localhost diskless-image]# cat /etc/shadow | grep root
root:$6$F2.LmNz/WbCyrWvu$kxtbWFKXAGlj9TigyNacmYcywWYD9fn7pjX7qOavSCTQVpPFBBmsyNo
zrWgUjc/WN352dxE3U9bksBZWxdVQJ0:16911:0:99999:7:::
[root@localhost diskless-image]#
```

然后切换到无盘系统，修改 shadow 文件中的 root 密码加密字符串。

```
[root@localhost diskless-image]# chroot /root/diskless-image/rootfs-mnt/
[root@localhost /]# vim /etc/shadow
[root@localhost /]# cat /etc/shadow | grep root
root:$6$F2.LmNz/WbCyrWvu$kxtbWFKXAGlj9TigyNacmYcywWYD9fn7pjX7qOavSCTQVpPFBBmsyNo
zrWgUjc/WN352dxE3U9bksBZWxdVQJ0:15980:0:99999:7:::
[root@localhost /]#
```

接着，创建网卡相关的配置文件。

```
[root@localhost /]# vim /etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE=eth0
BOOTPROTO=dhcp
ONBOOT=yes
```

之后，配置 dns 解析服务器。

```
[root@localhost /]# vim /etc/resolv.conf
nameserver [YOUR NAMESERVER IP]
```

配置 mtab 文件。

```
[root@localhost /]# vim /etc/mtab
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
devpts /dev/pts devpts rw,gid=5,mode=620 0 0
tmpfs /dev/shm tmpfs rw 0 0
none /proc/sys/fs/binfmt_misc binfmt_misc rw 0 0
```

配置 fstab 文件。

```
[root@localhost /]# vim /etc/fstab
tmpfs /dev/shm tmpfs defaults 0 0
devpts /dev/pts devpts gid=5,mode=620 0 0
sysfs /sys sysfs defaults 0 0
proc /proc proc defaults 0 0
```

通过如下命令安装一些常用库或者工具等。

```
[root@localhost /]# yum install libusb pciutils zlib libtomcrypt bind-utils
createrepo curl lsof
```

现在退出 chroot 模式，命令如下：

```
[root@localhost /]# exit
exit
[root@localhost diskless-image]#
```

卸载已经挂载的无盘镜像系统。

```
[root@localhost diskless-image]# umount rootfs-mnt/proc
[root@localhost diskless-image]# umount rootfs-mnt/dev
[root@localhost diskless-image]# umount rootfs-mnt
[root@localhost diskless-image]# df -h
Filesystem                Size  Used Avail Use% Mounted on
/dev/mapper/vg_livedvd-lv_root
                           18G   9.5G  7.5G   56% /
tmpfs                      751M    0  751M    0% /dev/shm
/dev/sda1                  477M   78M  374M   18% /boot
[root@localhost diskless-image]#
```

上述步骤就完成了镜像文件的制作，为了减少在启动过程中镜像传输的时间，可对镜像进行压缩，命令如下：

```
[root@localhost diskless-image]# tar zcvf rootfs.tgz rootfs.img
rootfs.img
[root@localhost diskless-image]#
```

8.4.3 测试无盘镜像

1) 将文件移动到相应 Cobbler 的文件夹中，命令如下：

```
[root@localhost diskless-image]# mkdir -p /var/www/cobbler/ks_mirror/diskless-centos6
[root@localhost diskless-image]# cp -r initrd vmlinuz rootfs.tgz /var/www/cobbler/ks_mirror/diskless-centos6/
```

2) 在 Cobbler 中添加 distro、profile 以及 system 信息。

以下命令用于添加 distro。

```
[root@localhost diskless-image]# cobbler distro add --name diskless-centos6
--kernel=/var/www/cobbler/ks_mirror/diskless-centos6/vmlinuz --initrd=/var/www/cobbler/ks_mirror/diskless-centos6/initrd
--kopts='ksdevice=bootif lang rw ip=dhcp root=live:http://192.168.88.110[注意替换为自己的IP]/cobbler/ks_mirror/diskless-centos6/rootfs.tgz nousb text ramdisk_size=2097152 kssendmac rd.writable.fsimg=1 rd.live.ram=1 rd.live.debug=1'
```

以下命令用于添加 profile。

```
[root@localhost diskless-image]# cobbler profile add --name diskless-centos6
--distro diskless-centos6
```

以下命令用于添加 system。

```
[root@localhost diskless-image]# cobbler system add --name centos6 --mac-address=08:00:27:F1:8C:CD[注意替换为自己的MAC] --interface=eth0 --ip-
```

```
address=192.168.88.119[注意替换为自己的IP] --gateway=192.168.88.1[注意替换为自己的网
网] --netmask=255.255.255.0[注意替换为自己的掩码]
```

可运行 Cobbler 的 sync 命令来使所有配置生效。

```
[root@localhost diskless-image]# cobbler sync
```

注意检查 rootfs.tgz 的文件权限，确保所有用户可读，命令如下：

```
[root@localhost diskless-image]# ll /var/www/cobbler/ks_mirror/diskless-centos6/
total 391292
-rwxr-xr-x 3 root root 29159963 Apr 20 14:56 initrd
-rw-r--r-- 1 root root 367296010 Apr 20 14:56 rootfs.tgz
-rwxr-xr-x 3 root root 4222448 Apr 20 14:56 vmlinuz
[root@localhost diskless-image]#
```

启动测试虚拟机，enjoy！

8.5 本章小结

本章主要介绍了系统的网络部署流程。运用 Cobbler 工具来实现网络部署的自动化及批量化，减少了系统管理员的重复工作量。在本章最后，简单地配置了一个基础的无盘系统，通过 Cobbler 来进行批量的部署。在今后日常工作中，无论是常用的有盘系统的安装部署，还是定制化的无盘系统的安装部署，都能通过使用自动化工具快速无误地完成。

Puppet 配置管理

本章将迎来 DevOps 的重头戏——Puppet，如果说自动化是运维效率的根本，那么 Puppet 就是自动化运维的神器，本章将通过一些简单明了的例子展开讲解，使大家快速入门。此外，本章还提供了大量深入的代码，方便大家高屋建瓴地了解 Puppet 的精华部分。诚然，我们经常听到一些运维抱怨自动化工具难学，还是批量脚本简单的说法，可是要知道，脚本是前期爽快，后期不仅容易出错，而且还不易于维护，谁用谁知道！所以，本章要打破自动化工具与运维之间的隔阂，让大家都可以看着显示器，悠闲地喝着咖啡。

9.1 什么是 Puppet

维基百科上说，Puppet 是一款开源的自动化配置管理工具，它可以运行在 unix-like 系统中，也可以运行在 Windows 中，并且使用了简单的声明式语言来抽象系统资源，来进行自动化管理。

通俗点讲，如果用户之前是使用一大堆 script 来进行服务器自动化运维的话，Puppet 就是一个已经定义好各种标准模块，只要声明并调用已有模块里的函数就可以完成自动化运维的工具。

再形象点讲，Puppet，英文意思“木偶”，作为系统管理员，只需要动动手里的小木棒，系统就像木偶一样，随意舞动，风姿摇曳。

9.1.1 Puppet 对于系统运维意味着什么

Puppet 也如其他自动化配置管理工具的信念一样，即一次投资，终生收益。它会把你

户在部署和配置服务时所修改的对象都抽象成一个资源，如下：

- ❑ File：配置文件的拷贝
- ❑ Cron：定期任务的部署
- ❑ Package：安装包的管理
- ❑ Service：服务的运行
- ❑ Exec：执行的 shell 命令

用户所要做的就是 Puppet master 上定义这些资源，完成后，在 Puppet agent 上就会定期自动做所有的事情，是的，从此以后安装服务再也不用登录机器了。

但是这是理想状态，现实情况是随着项目的发展，需要在 Puppet master 上作相应的调整，如更新配置文件、添加新的 cron job 等。用户会慢慢发现多数时间都是开着 Puppet master 的 terminal 在改 Puppet 代码。那么恭喜你，你已经迈入 DevOps 的大堂了。

9.1.2 为什么选择 Puppet

1. 比较

提到 Puppet，肯定有人会想到其他类似工具，老牌一点的如 cfengine、bladelogic（商业），新潮点的如 chef、ansible、salt。这些工具各有特点，如果对这些技术感兴趣，可以使用 google trends 查看各工具的流行指数，github 的指标（星级、commit 和 folk 数，等等）查看该工具的社区活跃度，当然也可以参考百度指数。

本章讲解 Puppet，这也说明了笔者的选择。笔者属于比较懒的一类“攻城狮”，不仅不愿意尝试新鲜事物，而且懒得为了生产服务频繁地线上维护，尤其像自动化配置管理工具这样的关键性应用，不得不谨慎小心，维护之前必须要写好详尽的 plan 和 roll back plan，真是费力伤神的事情，因此笔者在关键服务上更倾向于选择不太陈旧的成熟方案，在开放环境和非关键服务上选择新潮点的方案。Puppet 从 2005 年成立至今，不仅有活跃的社区，数千成熟的模块，而且还获得了上亿美元的投资，建立了商业 support，整个生态圈是相当健康的。

2. Puppet 的起源

笔者选择它的另外一个重要的原因是 Puppet 的作者 Luke Kanies 的有趣故事。Luke 是 Puppet 的作者、创始人以及 CEO，三重光环加身，经历确实丰富多彩，老牌点的说法是“故天将降大任于斯人也，必先苦其心志，劳其筋骨，饿其体肤，空乏其身”，新潮点的说法是一个“X 丝的打怪升级之路”。

Luke 的童年时光是在田纳西的一个农场上度过的，用他创业时接受采访的原话来说，“我的童年是在一个有 1600 人的嬉皮士公社渡过的，”他笑了笑，“现在我有一个几百万美金的公司，而当时我直到 8 岁还没有一个厕所。”这造就了 Luke 斗士精神，以及后来他为人处事的态度。

到了 1992 年，Luke 屁颠屁颠地跑到威斯康辛州的一所名不见经传的 Northland College

学起了化学，过了一年 Luke 同学发奋图强去了全美排名前 50 的 Reed college，和乔布斯（乔帮主）成为了校友。可是这条路并不平坦，该大学课程犹如国内大学一样，不仅要学习希腊及罗马的古典文化，还要在四个拓展领域选课：文学、哲学、宗教、艺术等领域；历史、社科、心理学等领域；自然科学等领域；数学、逻辑、语言学或外语等领域，可谓德智体美劳全面发展（事实证明，国内模式也能出乔布斯这样的大拿）。到了大四，Luke 终于交出了满意答卷“Site-directed Mutagenesis in Soy Cytosolic Ascorbate Peroxidase”，这论文题目，有兴趣的同学可以自己翻译，他终于要踏入社会了。

到了 1997 年，Luke 带着化学天赋懵懂地踏入社会，做了一年不到的 mac sysadmin 和两年不到的 call center 装机工，终于跑到一家叫 bluestar 的公司正儿八经当起了 system engineer，用他的话来说“didn't want to fix computers forever！”（我再也不要修电脑啦！），多么质朴的话语，道出了我们众多“攻城狮”的心声。在两年的磨砺中，他终于成了脚本小子，实现了脚本化自动化运维。

2001 年，不安分的 Luke 又觉得什么都要用脚本写实在太麻烦了，他开始研究配置管理工具的鼻祖 cfengine，并且跑到一家叫 Caterpillar 的融资公司担任顾问，同年又成立了 Reductive Consulting 的咨询公司（其实和大多数有经验的“攻城狮”接私活是一个性质）。在接下来三年里，Luke 深入研究了 cfengine，并积极地改进并贡献代码。但是随着时间的推移，他发现 cfengine 的生态圈并不好，大家都不愿意分享模块代码，他感到非常失望，并开始需求其他解决方案。

到了 2004 年，斗士 Luke 来到了大名鼎鼎的商业软件 bladelogic 担任产品设计，虽然只待了 7 个月，虽然他曾抱怨 bladelogic 对初创项目并不友好，但是正是 bladelogic，促使他萌发了做一个开源解决方案的想法。

于是，到了 2005 年，他和老婆商量再三后，准备正式单飞，把咨询公司变成一个真正的软件公司，也就是后来的 Puppetlabs。关于 Puppet 是如何发展的，社区运作是否健康，网上随便一搜都有，这里不再赘述。只想提一句，成功的男人背后必定有一个默默支持他的女人，Luke 在他老婆怀孕的那一年，飞行里程是 9 万英里（赤道上大约绕三圈半），就连他老婆生孩子的那一刻他还在飞机上，最后顺利产下一对双胞胎，真是人生赢家。

好了，说了这么多，其实想表达的观点是，Luke 的经历很对笔者的口味，当然有可能有人还能从中找到自己的影子，但这些并不是当初我们的项目选择 Puppet 的原因，接下来我们就一起走入 Puppet 的世界，欣赏 Puppet 之美。

9.2 安装 Puppet

9.2.1 准备工作

第一步当然是选择系统，最常见的是选择 CentOS 6 作为环境来安装。不过接下来要说的并不是常规的准备工作，而是有关 Puppet 的准备工作。

1. 选择 Puppet 模式


Puppet 具有 2 种模式: master 和 masterless。顾名思义, master 是以传统的 C/S 模式运行, 每台机器都会跑一个的 Puppet agent, 而 masterless 就是预先把 Puppet 代码拷贝到每台机器上, 像一个脚本一样独立运行, 接下来进行具体分析。

(1) Agent/Master 模式

大多数情况下, 推荐用户选择这种模式, 集中化管理 Puppet 代码, 并可以根据一些现成的工具, 通过读取每台 agent 传到 master 的 report, 来了解 Puppet 整体运行状态, 如 Puppet dashboard。可以这么说, 如果你的项目是以下情景:

- ☐ 是一个 agent 半小时同步一次 master 就足够的环境。
- ☐ agent 少于 1000 台。
- ☐ 有一台不是很差的空闲服务器, 8GB, 4 核, 带 RAID 卡与电池的一台 Dell 入门级 server。

那么恭喜你, 只需要所写的 Puppet 的代码不是太烂, 你就可以毫不犹豫地选择 agent/master 模式。

 **说明** 默认的 master 是通过独立的 daemon 运行的 (官方称 WEBrick 方式, 是 Ruby 自带的简易 http 库), 性能不是特别好, 胜在简单且足以应对测试环境和数十台 agent, 如果正式投入运营, 推荐使用 Apache+Passenger 的方式, 以获得更好的性能。


此外, 官方还有一个更新的架构叫 Puppetserver, 目前是 1.0.8 版本, 用 JRuby (一个采用纯 Java 实现的 Ruby 解释器) 编写, 虽然笔者不喜欢 Java, 但是官方推崇 Java 的原因是遇到了性能瓶颈, 靠 Ruby 是无法简单解决的, 于 2014 年圣诞节前夕刚进入 1.0.0 版本, 建议继续观望。

(2) Masterless 模式

Puppet 官方称 standalone 模式, 使用 Puppet apply 来执行本地 Puppet 代码, 这种模式是一种极端模式, 如果项目是以下几种情景那么可以试用。

- ☐ 只有两三台服务器 (其实两三台机器更适合用纯 shell 脚本)。
- ☐ 10 000 台服务器。
- ☐ 1000 台机器需要在 1 分钟内全部跑一遍 agent 到 server 的同步。

当然后面两种情况还是可靠适当的调优和堆机器来解决的, 只要将 master 之间的同步做好即可, 相信这么大的项目 10 台左右服务器还是值得投入的。事实上, 在进行一定的后期调整后, master 可以做得和 masterless 一样好, 并且节省了硬件开销, 加快了 agent 的运行速度, 笔者的项目就曾经评估和实验过这个方案, 后来比较了维护成本和新人学习成本, 还是选择了 master 模式, 相应的投入还是值得的。

 **说明** 本地 masterless Puppet 的运行其实可以试用于新机房中第一台 master 服务的搭建,

一般 2 个机房之间的网络打通要在项目中后期才会实现，提早完成第一台 master 服务的搭建，可以加快项目交付的进度。

2. 选择 Ruby 环境

官方建议以下三个版本：

- ☐ Ruby 2.0.x
- ☐ Ruby 1.9.3
- ☐ Ruby 1.8.7

原因很简单，官方是基于这三个 Ruby 版本做测试的。而 Centos 6 上默认就是 Ruby 1.8.7，如果用户使用的是 CentOS 5，那么很不幸，需要到网上找一个非官方 Ruby 1.8.7 的 el5 rpm 包，或者自己编译安装。笔者的项目中有部分 CentOS 5 的机器，碰到过坑，即 Ruby 1.8.5 跑 Puppet agent daemon 的时候会有内存持续溢出，虽然可以靠 cron 定期跑 service Puppet restart 解决，但最终还是自行编了一个 Ruby 1.8.7 的 rpm 包来解决。

3. 检查网络配置

(1) 防火墙

Puppet master 默认使用 8140 端口，所以建议检查硬件防火墙以及本地 iptables，如果对 iptables 不熟悉，可以使用 service iptables stop 来关闭它。

(2) 域名解析

Puppet master 和 agent 之间的交互是通过域名实现的，保证两者之间的域名解析正常是非常重要的。

master 的配置

在有 DNS Server 的时候，只需为 Puppet master ip 预先加一条 DNS A 记录解析即可（解析域名为 puppet，有 FQDN，即是 puppet.your_domain.com）。如果没有 DNS Server，需要在每台 agent 中的 hosts 里加入 master ip，用于解析。在搭建工作中，这将为第一次接触 Puppet 的读者，大大减少可能会碰到的麻烦。

agent 的配置

由于 Puppet master 与 agent 是用 SSL 来进行签名加密传输的，所以要满足：

- ☐ 每台 agent 必须要有独立的 hostname。
- ☐ 每台 agent 都可以正常解析 Puppet master 的 IP 为域名“puppet”。

因此，如果你有 DNS，需要保证每个 agent 的正向和反向解析都正常，即有 A 记录，也要有 PTR 记录，如果没有 DNS，则不仅需要在 Puppet master 的 hosts 里加入每个 agent 的 IP，也要在每台 agent 的 hosts 里加入 agent 的本机 IP 和 hostname 作为解析。



说明 本文将不再展开如何搭建 DNS，同时，强烈建议读者使用 DNS，管理成百甚至上千台服务器，如果没有用 DNS 的这样一个好习惯，平时的运维可能会遇到无数的烦恼和陷阱。

4. 检查时间同步

通常 NTP 最容易导致的问题是 SSL 证书不合法, 该证书会被认为是过期或者是未来的证书, 之前已提到 Puppet master 与 agent 是用 SSL 来进行签名加密传输的, 所以时间同步是用户应该预先检查的问题。

5. 选择 Puppet 版本

最后, 也是最重要的, 选择 Puppet 版本。

目前最新版本是 4.*.* 系列, 是用 JRuby 写的 Puppetserver, 开源项目当然最欢迎小白鼠, 不过对于新手来说, 还是推荐稳定版本, 即 3.*.* 系列, 本书使用的是 Puppet 3.8.*, 因为它有如下特性:

- ☐ 性能更好
- ☐ 入手更简单

当然目前还有 2.*.* 系列的版本, 很多国内的文档都是基于此版本, 不过笔者认为该版本实在跟不上时代, 建议用户丢掉手中如何撰写第一模块的其他资料吧, 把奇怪的 import, 用 genmanifest 来产生第一个天书一般的 site.pp, 还有新手完全看不懂的默认变量, 这些统统卸掉。然后, 根据本书中的 site.pp 章节来开始你轻松愉快的 Puppet 之旅吧!

9.2.2 安装一个服务端

1) 导入官方 repo, 命令如下:

```
[root@puppet /]# rpm -ivh https://yum.puppetlabs.com/puppetlabs-release-el-6.noarch.rpm
```

2) 安装 puppet-server, 命令如下:

```
[root@puppet /]# yum install puppet-server
```

3) 删除默认证书, 命令如下:

```
[root@puppet /]# puppet cert clean --all
```

4) 产生 dns name 为 “puppet” 的证书, 命令如下:

```
[root@puppet /]# puppet cert generate puppet --dns_alt_names puppet
```

9.2.3 安装一个客户端

第一步, 导入官方 repo, 命令如下:

```
[root@agent /]# rpm -ivh https://yum.puppetlabs.com/puppetlabs-release-el-6.noarch.rpm
```

第二步, 安装 Puppet agent, 命令如下:

```
[root@agent /]# yum install puppet
```

9.2.4 连接第一个客户端

首先，检查 Puppet server 是否可达客户端，命令如下：

```
[root@agent ~]# ping puppet
[root@agent ~]# telnet puppet 8140
```

如果上述命令失败，需要排错后，再执行后续步骤。

首次请求 Puppet server 的命令如下：

```
[root@agent ~]# puppet agent -t
Info: Creating a new SSL key for agent.example.com
Info: Caching certificate for ca
Info: csr_attributes file loading from /etc/puppet/csr_attributes.yaml
Info: Creating a new SSL certificate request for agent.example.com
Info: Certificate Request fingerprint (SHA256): 4A:6B:B9:83:F1:83:B7:18:09:96:8A
:73:9A:62:65:C4:BE:E7:EF:C6:31:EF:91:85:62:93:73:75:66:F8:4C:8F
Info: Caching certificate for ca
Exiting; no certificate found and waitforcert is disabled
```

可以看出，第一次 agent 已自生成证书，并发送给 master 等待认证。

然后在 Puppet server 上查看认证请求并接受签名，命令如下：

```
[root@puppet ~]# puppet cert list
"agent.example.com" (SHA256) 4A:6B:B9:83:F1:83:B7:18:09:96:8A:73:9A:62:65:C4
:BE:E7:EF:C6:31:EF:91:85:62:93:73:75:66:F8:4C:8F

[root@puppet ~]# puppet cert sign agent.example.com
```

如果失败请再次检查准备工作的网络配置部分。

之后返回 agent，再次执行上面的命令：

```
[root@agent ~]# puppet agent -t
Info: Caching certificate for agent.example.com
Info: Caching certificate_revocation_list for ca
Info: Caching certificate for agent.example.com
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for f1d7e51bf943.example.com
Info: Applying configuration version '1428845851'
Info: Creating state file /var/lib/puppet/state/state.yaml
Notice: Finished catalog run in 0.23 seconds
```

最终，agent 与 master 成功互联，运行成功！

9.2.5 Puppet master 上的 site.pp

site.pp，正如大多数文章所说，它就是 Puppet agent 运行时首先要执行的第一段 Puppet 代码。其默认路径为：

```
[root@puppet /]# puppet master --configprint all | grep '^manifest '
manifest = /etc/puppet/manifests/site.pp
```

它具有如下功能。

1) 能直接写 Puppet 代码，全局管理 resources。


在 master 上加入如下代码：

```
[root@puppet /]# vim /etc/puppet/manifests/site.pp
file { '/tmp/oxx':
    content => 'test23',
    owner => root,
    group => root,
    mode => 0440,
}
```

在 agent 上执行如下命令：

```
[root@agent /]# puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Info: Caching catalog for 98876ed2a207.example.com
Info: Applying configuration version '1429452948'
Notice: /Stage[main]/Main/File[/tmp/oxx]/ensure: defined content as '{md5}b48cca5aebb82a328227b78d899506f5'
Notice: Finished catalog run in 0.46 seconds
```

此时会发现，agent 上自动创建了一个 /tmp/oxx 的文件，文件内容、属性和所定义的一致。

 **注意** 这里暂不展开讲 Puppet 语法，先尽量给出一些浅显易懂的例子，让大家对 Puppet 语法有感观上的良好体验后，再在后文具体展开，相信会更加直观，类似于小孩学习语言都是从依样画葫芦开始的。

2) 能定义 node，使用不同的 Puppet 代码，管理 resources。

在 master 上加入如下代码：

```
[root@puppet /]# vim /etc/puppet/manifests/site.pp
node agent.example.com {
    file { '/tmp/oxx':
        content => 'test23',
        owner => root,
        group => root,
        mode => 0440,
    }
}
```

会发现该 file 的 resource，只能在 agent.example.com 上应用。



node default 是默认规则，所有 agent 都会主动应用代码里的 resources，也就是大括号 ({}) 里的资源，推荐以这种方式来定义全局 resources。

3) 能定义 class，在 class 里面定义 resources，让不同的 node 使用不同的 class 中的 resources。

在 master 上加入如下代码：

```
[root@puppet /]# vim /etc/puppet/manifests/site.pp
node agent.example.com {
    include example
}

class example {
    file { ['/tmp/ooxx':
        content => 'test23',
        owner => root,
        group => root,
        mode => 0440,
    ]
}
```

会发现这样写的好处是，class 可以重复利用，另外一个 node 如果也想要应用 example class，只需要写一个 include example 即可。

4) 还可以 import 其他 pp 文件。

很多文章都会这么写，在 /etc/puppet/manifests/site.pp 加入如下内容。

单独放一大堆 pp 到当前 nodes 这个目录下，让 site.pp 读取并应用其中的代码。

```
import "nodes/*.pp"
```

在 modules.pp 里定义各个模块的路径。

```
import "modules.pp"
```


如果用户之前用过，或者准备这么用，请立刻停止，因为新版的 Puppet 已经把 import 默认为 deprecated，也就是弃用的状态，理由就是不需要。

对于 import "nodes/*.pp"，官方认为，nodes 由于可以支持正则，所以不需要大量的 pp 文件来支持，正如文章前面说过的，如果真想管好成百上千台机器，使用 DNS 吧，因为可以使用如下几种代码方式定义 node。

```
node 'www1.example.com', 'www2.example.com', 'www3.example.com' {
    ...
}

node /^www\d+$/ {
    ...
}
```

```
node /^(foo|bar)\.example\.com$/ {
  ...
}
```

 **说明** 由于 Puppet 的 regex 来源于 Ruby，更多语法请参考 <http://ruby-doc.org/core-2.1.1/Regexp.html>。

如果以上方式还无法满足用户需求，那么请参考后文 ENC 的管理方式。

对于 import "modules.pp"，官方认为，modules 就应该放在标准路径，建议就放在 /etc/puppet/modules 里，这更符合 Linux 的习惯。代码如下：

```
[root@puppet /]# puppet master --configprint all | grep '^modulepath '
modulepath = /etc/puppet/modules:/usr/share/puppet/modules
```

综上所述，只要学习了如何使用 site.pp，就能进行 agent 的 resources 管理，然而还是建议采用以下方式进行最干净的管理。

- ❑ site.pp 就应该只包含 node 的定义和 include 中的 class，不包括 resources 管理代码。
- ❑ resources 只定义在 class 里，而 class 的完整定义应该放在 /etc/puppet/modules 中。
- ❑ 停止使用 import，已经不需要了。

理由很简单，你应该避免上千行代码在一个文件里面，就如 Linux 不会建议把所有开机启动程序放在 /etc/rc.local 里，而是每个程序都有相应的 /etc/init.d 脚本一样，用 runlevel 来控制它们，在 site.pp 中，尽量只定义不同机器（node）应该运行哪个 Puppet 模块即可，具体逻辑放在相应的 Puppet 模块中。

 **说明** 如果后续学习并使用了 enc(External Node Classifiers)，完全可以扔掉 site.pp。

9.2.6 制作第一个模块

上节讲了如何用最佳方式干净地管理 Puppet，现在来着手制作第一个模块。

1. 语法

在制作模块之前，大家要问的第一个问题肯定是：Puppet 是什么语法？虽然笔者之前提到过多读代码产生的感觉比语法更重要，但是为了做一个参考，方便后文快速展开，这里还是提一提。

还是以上文的一个文件管理为例。单一 resources 的代码如下：

```
file { ['/tmp/ooxx':
  content => 'test23',
  owner => root,
  group => root,
  mode => 0440,
}
```

这里要先说明一下关于资源的几个概念。

- ❑ **resource type** : 资源种类, 这里使用了“file”, 这个 Puppet 内置的一个最常见的种类。
- ❑ **resource title**: 资源的标题, 这里命名为“/tmp/ooxx”, 在“file”这种 type 里为文件的绝对路径。
- ❑ **resource attribute** : 资源的属性, 这里有四个属性, 分别是 content、owner、group、mode, 相信属性名已经清楚地说明了它们的含义, 不需要过多的解释。

代码中标点符号的用法如下:

- ❑ 冒号 (:), 是紧挨着 title 之后的。
- ❑ 单引号 ("), 要对所有自定义的 string 使用。
- ❑ 箭头号 (=>), 左边为 attribute 的名字, 右边为该 attribute 所要赋予的值。
- ❑ 逗号 (,), 每个 attribute 定义好之后需要加上逗号, 这里最后一个 attribute 建议也加上, 因为实际操作过程中, 用户都喜欢复制上一行来写一个新的 attribute, 而往往都会忘记补上这个逗号。
- ❑ 大括号 ({}), 包裹了一个 resource type 里的所有定义。

最后一点, 以上都是英文标点。

所以, 提炼后的语法形式如下:

```
type {'title':
  attribute1 => value1,
  attribute2 => value2,
}
```

对于多个 resources 组成一个类的代码, 形式如下:

```
class ntp_client {
  file { ['/tmp/ooxx':
    content => 'test23',
    owner   => root,
    group   => root,
    mode    => 0440,
  ]

  package { 'ntp':
    ensure => installed,
  }

  service { 'ntp':
    name     => 'ntpd',
    ensure   => running,
  }
}
```

同样解释一下其中的几个概念和标点符号的用法。

- ❑ `class`: 类的定义, `ntp_client` 是类名。
 - ❑ `resources`: 该类中包括了 `file/package/service` 三个 `resources`, 分别以空行相隔。
 - ❑ `{}`: 大括号来包裹所有三个 `resources` 的定义。
- 最后在 `site.pp` 中 `include` 即可。

2. 构思模块

现在就从一个简单的需求出发来制作一个简单的模块, 比如想要在登录系统后, 使用 `motd` 告诉来访者一些基础的信息。假设有如下需求:

- ❑ 显示系统基础信息。
- ❑ 安装 `dstat` 来抓取各种 `stat`。
- ❑ 显示 Puppet 相关信息。
- ❑ 把以上需求做成一个脚本。
- ❑ 需要有 `cron` 来定期执行这个脚本, 去更新 `/etc/motd`。

将这些需求转义成 Puppet, 则为:

- ❑ 用一个 `package resource type` 来安装 `dstat`。
- ❑ 用一个 `file resource type` 来放抓取脚本, 该脚本会自动更新 `/etc/motd`。
- ❑ 用一个 `file resource type` 来放 `cron` 的配置。
- ❑ 用一个 `service resource type` 来保证 `crond` 的运行。


3. 撰写模块

这里不展开完整的模块目录结构, 就从最这个最简单的例子开始。

第一个 `motd` 模块的目录结构如下:

```
[root@puppet /]# find /etc/puppet/modules/motd/
/etc/puppet/modules/motd/
/etc/puppet/modules/motd/fact.d
/etc/puppet/modules/motd/lib
/etc/puppet/modules/motd/files
/etc/puppet/modules/motd/files/generate_motd.sh
/etc/puppet/modules/motd/files/motd_cron_config
/etc/puppet/modules/motd/manifests
/etc/puppet/modules/motd/manifests/init.pp
```

可以看到, 在 `/etc/puppet/modules/` 下创建了 `manifests` 目录来存放 `init.pp` 清单文件, 并且创建了 `files` 目录来存放 `motd_cron_config`、`generate_motd.sh` 这两个文件。

 **注意** 在目前的版本, `modules` 里至少有一个 `module` 要包含 `fact.d` 和 `lib` 目录, 即使没有 `custom` 的 `fact` 和 `lib`。

第一个 `motd` 模块的 `manifests/init.pp` 如下:

```
class motd {
```



```

file { ['/usr/local/bin/generate_motd.sh']:
    source => puppet:///modules/motd/generate_motd.sh,
    owner => root,
    group => root,
    mode => 0775,
}

file { ['/etc/cron.d/motd_cron']:
    source => puppet:///modules/motd/motd_cron,
    owner => root,
    group => root,
    mode => 0644,
    require => Package['cronie'],
}

package { ['dstat', 'cronie']:
    ensure => installed,
}

service { 'cron':
    name      => 'crond',
    ensure    => running,
    require   => Package['cronie'],
}

```

在上述代码中，新 resources、package 和 service 的具体用法后文有详解，现在一笔带过。对于 ['dstat', 'cronie']，这里用了个小技巧，resources 中的 title 可以使用 array 形式声明。

source => puppet:///modules/motd/generate_motd.sh 是 file 中最常见的一种用法，该条 attribute 的意思是，到 Puppet 服务器上 :///modules 目录下 /motd 这个模块下 / 从 files 下中找 generate_motd.sh 为源文件。要说明的是，source 的定义中特地省略了 files 的显式定义。

此外，require 的意思是应用这个 resource 之前，需要依赖 resource Package['cronie'] 的安装。在 require 语句中，所依赖的 resource 的 type 首字母要大写。

第一个 motd 模块的 files/generate_motd.sh 如下：

```

#!/bin/bash

if [ -f '/var/lib/puppet/state/state.yaml' ]; then
    PUPPET_LAST_RUN="Last Run date - `stat /var/lib/puppet/state/state.yaml | awk
    '/Modify/ {print $2,$3}'`"
else
    PUPPET_LAST_RUN="Never Run on this server"
fi

MOTD="
+++++++ System Data :+++++++
+ Hostname = `hostname`

```

```
+ Address = `ifconfig eth0 | awk -F':| *' '/inet addr/ {print $4}`
+ Kernel = `uname -r`
+ Uptime = `uptime | sed 's/.*up ([^,]*) , .*1/'`
+ CPU = `cat /proc/cpuinfo | awk -F: '/model name/{print $2}' | head -1`
+ Memory = `cat /proc/meminfo | awk '/MemTotal/ {print $2}'` kB
+++++: Dstat :+++++
`dstat -a 1 3`
+++++: Puppet :+++++
`puppet agent --configprint all | grep '^server '`
`facter | grep -E '^rubyversion|^puppetversion '`
$PUPPET_LAST_RUN
"
echo "$MOTD" >/etc/motd
```

第一个 motd 模块的 files/motd_cron 如下：

```
*/5 * * * * root bash /usr/local/bin/generate_motd.sh
```

第一个 motd 模块在 site.pp 中定义了哪个 agent 会应用它，代码如下：

```
node agent.example.com {
    include motd
}
```

接下来可以在 agent 上执行 puppet agent -t 看结果了。

9.3 深入 Puppet

9.3.1 深入 resources type

本节用于深入了解各种常用的内建 resource type，因为定义各种 type 是 Puppet 的灵魂，任何 module 都是由多个 resources 组成的，所以这里有必要展开来了解。

一般而言，内建的 resource type 都包含如下三个方面。

- attributes: 属性，比如 file type 里的 owner、group、mode 等。
- providers: 提供者，比如 package type 里的 yum、apt、pip 等。
- features: 提供者所具有的特性，比如 package type 里，yum 和 apt 的 ensure attribute 都有 purge 的功能，而 rpm 却没有。

这里主要是讲述 resources 的 attribute 的用法和使用场景，至于 provider 和 features，Puppet 会根据系统选择默认的 provider，所以一般情况下，只需要关心如何个性化属性 (attribute) 即可。

1. package 资源的配置管理

(1) name

name 为包名，默认值为 resource title，与定义的标题一致。使用频率中等。以下是两

个场景中的使用。

场景一，在不同机器有不同变量来定义包版本的时候使用。

在 `site.pp` 中，不同 `node` 定义不同的变量，然后 `name` 使用变量来实现，`site.pp` 中的代码如下：

```
node agent1.example.com {
    $dstat_pkg = dstat-0.7.0-1.noarch
    include motd
}

node agent2.example.com {
    $dstat_pkg = dstat-0.6.9-2.noarch
    include motd
}
```

`motd/manifests/init.pp` 中的代码如下：

```
package { 'dstat':
    name => $dstat_pkg
    ensure => installed,
}
```

场景二，用于 `package` 替换，但无需改 Puppet 代码。比如在实际应用当中，想在一些环境内测试一些 `package` 的替代品，但又不想更改 Puppet，这时候就可以做如下 tricks。

`site.pp` 中的代码如下：

```
node agent1.example.com {
    $dstat_pkg = atop # 我们发现atop是一款相当有趣的带有history的top like工具，可以完美替换dstat
    include motd
}

node agent2.example.com {
    $dstat_pkg = dstat-0.6.9-2.noarch
    include motd
}
```

`motd/manifests/init.pp` 中的代码如下：

```
package { 'dstat':
    name => $dstat_pkg
    ensure => installed,
}
```

(2) ensure

`ensure` 用于确保 `package` 在操作系统上的状态，默认值为 `installed` 状态，即确保 `package` 是已安装的状态。以下是其他可选值。

❑ `present`：等同于 `installed`。

- ❑ **absent**: 卸载状态, 在 CentOS 中, 如果有别的 package 依赖的话, 报错。
- ❑ **purged**: 卸载状态, 在 CentOS 中, 如果有别的 package 依赖的话, 一并卸载。
- ❑ **latest**: 一直保持安装为最新版本。

(3) install_options

install_options 是安装参数。顾名思义, 在 CentOS 中, 就是指定 yum install 的参数, 使用场景较少。

(4) provider

provider 是安装的来源工具。之前提过, 在 CentOS 中, 默认 provider 是 yum, 其实还可以使用 rpm, 甚至是 pip、gem 来管理 Python 和 Ruby 模块包, 这里不具体展开。有兴趣的可以结合 **install_options**, 做出适用于 production 的代码语言环境。

2. file 配置管理

上文讲了比较简单的 package 资源, 现在来讲一下另外一个常见 resource 的重头戏, file。

(1) path

path 为 file 路径, 默认为 title 中指定的路径, 使用场景少。因为 title 中已具备使用路径的功能, 所以一般不用该 attribute, 下面针对一个特殊场景给出正反两种示例。

不够优雅的用法如下:

```
file { ['/usr/local/bin/jack_script1.sh', '/usr/local/bin/jack_script2.sh', '/usr/local/bin/jack_script3.sh']:
    mode => '755',
    user  => 'jack',
    group => 'jack',
}
```

以下则是使用 **path** 后优雅的用法:

```
file { ['jack_script1.sh', 'jack_script2.sh', 'jack_script3.sh']:
    path => '/usr/local/bin/',
    mode => '755',
    user  => 'jack',
    group => 'jack',
}
```

(2) ensure

ensure 用于确保文件在系统上的状态。与 package 不同的是, **ensure** 在 file 中没有默认值。file 中 **ensure** 可以接受如下值。

- ❑ **present**: 代表存在, 即使后文并未指定该 file 的内容, Puppet 也会为其创建一个空文件。
- ❑ **absent**: 与 present 相反, 即使原先存在, Puppet 也会主动删除已存在的文件。
- ❑ **file**: 这个值会确保目标文件是一个普通文件, 不是 link、block 等其他文件类型。

- ❑ **directory**: 这个值会确保目标是一个目录。
- ❑ **link**: 即创建一个 link, 需要与 **target** 连用, 指定要 link 到哪一目标文件。

使用示例如下:

```
file { ['/tmp/dsl']:
  ensure => link,
  owner  => root,
  group  => root,
  mode   => 0644,
  target => '/etc/passwd'
}
```

(3) content

content 用于确保文件的内容。它是第一种 Puppet 管理文件内容的模式, 它接受 **string**, 也可以接受 **file** 和 **template** 的 **function** 的 **string** 返回值, 又或者自定义 **function** 的 **string** 返回值。

以下是 **string** 的示例代码:

```
$hello_msg = "hello world"

file { ['/tmp/dsl']:
  ensure => present,
  owner  => root,
  group  => root,
  mode   => 0644,
  content => "$hello_msg"
}
```

要说明的是:

- ❑ Puppet 里面变量定义和声明都要加 \$, 这个语言风格对于新手来说是很友好的。
- ❑ \$hello_msg 末尾是没有空行的, 要么加入 \n, 要么定义的时候在最后一个 " 前空一行, 这个空行对于某些配置文件来说是必须的, 比如 /etc/cron.d/ 里面的文件。

下面来看 **function** 返回的 **string** 值。

1) 首先, 是内建的 **function** 和 **file()**, 这是前文提到的另外一种文件内容管理方式, 示例代码如下:

```
file { ['/etc/cron.d/motd_cron']:
  source => puppet:///modules/motd/motd_cron,
  owner  => root,
  group  => root,
  mode   => 0644,
  require => Package['cronie'],
}
```

上面代码的效果等价于如下代码:

```
file { ['/etc/cron.d/motd_cron':
```

```

content => file("motd/motd_cron"),
owner => root,
group => root,
mode => 0644,
require => Package['cronie'],
}

```

注意，这里的路径：

```

source => puppet:///modules/motd/motd_cron,
content => file("motd/motd_cron"),

```

都会去找 /etc/puppet/modules/motd/files/motd_cron。

肯定有读者会问，这两种到底有什么区别呢？file function 似乎更简洁，而且可以用 template 和自定义 function，标准统一多好啊。下面简单地说下 source 存在的理由。

- ❑ 性能方面：file function 其实是把文件内容都以 catalog 的形式传给了 agent，它在 agent 上的执行相当高效，牺牲的是 master compile 所带来的开销。
- ❑ 扩展性方面：source 指定的方式是一个 URI，可以通过 fileservers 的 Puppet master 参数来搭建多个 file servers，以满足大型集群的需要。
- ❑ 递归性方面：source 可以指定一个目录，来递归的应用到目标机器上，详情见下文 recurse 的例子。

2) 其次，是 template 的内建 function()。template 也是 file 的核心内容，如果说在 Puppet 搭建好之后，80% 的时间我们在维护的 Puppet 代码是 file resource，那么这其中的 80% 又都是在维护 template 来让代码更具通用性、灵活性。

site.pp 中的代码如下：

```

node agent1.example.com {
    $dstat_pkg = atop # atop是一款相当有趣的带有history的top like工具，可以完美替换
                    dstat
    include motd
}

```

motd/manifests/init.pp 中的代码如下：

```

file { ['/etc/cron.d/motd_cron':
    content => template("motd/motd_cron.erb"),
    owner => root,
    group => root,
    mode => 0644,
    require => Package['cronie'],
}

```

motd/templates/motd_cron.erb 中的代码如下：

```

*/5 * * * * root bash /usr/local/bin/generate_motd.sh && echo "I installed <%= @
apache_pkg -%>" >>/etc/motd

```

在 agent 中可以看到替换变量后的完整文件内容，内容如下：

```
[root@agent1 /]# cat /etc/cron.d/motd_cron
*/5 * * * * root bash /usr/local/bin/generate_motd.sh && echo "I installed atop"
>>/etc/motd
```

这里说明一下上述例子中 template 使用到的一些语法和概念。

- ❑ erb：在 Puppet 中称 template 文件为 erb 文件。其实，这也是 Ruby 中内建的 template 标准库。
- ❑ <%= -%> 当中包含了变量，其实，这里也可以加入任何 Ruby 代码，后文会展开。
- ❑ @ 在 template 中用于引用变量。
- ❑ 变量定义的位置，变量在 Puppet 中可以定义于许多位置，可以是 site.pp，也可以是 manifests/init.pp，还可以是 enc、facter、自定义 function 等任何 Puppet 会运行到的地方。下面给出一个 tricks 可以知道到底 Puppet 在运行的时候会在每个 agent 端产生哪些变量供 template 里调用。

```
file { ["/tmp/facts.yaml":
  content => inline_template("<%= scope.to_hash.reject { |k,v| !( k.is_
    a?(String) && v.is_a?(String) ) }.to_yaml %>")
}
```

更多与 template function 和自定义 function 的相关知识会在后文中详解。

(4) group/owner/mode

group/owner/mode 用于确保文件的属性。熟悉 Linux 文件属性的读者应该知道它的含义，这里不再赘述。

(5) replace

replace 表示需不需要覆盖文件内容，默认是需要。设置成 false 可以让 Puppet 只会管理新文件的初始化。对于老文件，不覆盖其内容，只管理其他属性，如文件权限。例如想把 SSH 的 authorized_keys 推到所有机器上方便管理，但是又不确定是否有其他管理员已经在部分机器上做了这件事，且还有其他 keys，比如中央备份服务器的 keys，那么使用 replace 语法即可以实现该场景的需求。

(6) validate_cmd

validate_cmd 用于覆盖前面的检查命令。通常是用来验证配置文件的正确性的，下面是一个简单的 Apache 配置文件的例子。

```
file { ['/etc/httpd/conf/httpd.conf':
  content      => 'I am a bad configuration',
  validate_cmd => '/usr/sbin/httpd -t -f %',
}
```

注意，这里的 % 代表了 title 的名字 '/etc/httpd/conf/httpd.conf'，所以这里的 title 名字

一定是绝对路径。

(7) source

source 指源文件，即准备应用到 agent 的静态文件。关于 source，前文讲过，应该遵循 puppet:///modules/name_of_module/filename 规范来指定路径，并且在 /etc/puppet/modules/name_of_module/files/filename 下存放源文件。而关于 source 另外一个重要特性递归，会在下文着重展开。

现在先来说说另外一个有趣的特性，多个 source 的定义，以及它的意义。

site.pp 中的代码如下：

```
node agent1.example.com {
    $dstat_pkg = atop # atop是一款相当有趣的带有history的top like工具，可以完美替换dstat
    include motd
}
```

motd/manifests/init.pp 中的代码如下：

```
file { ['/etc/cron.d/motd_cron':
    source => [
        "puppet:///modules/motd/motd_cron.$host",
        "puppet:///modules/motd/motd_cron.$dstat_pkg",
        "puppet:///modules/motd/motd_cron",
    ]
    owner => root,
    group => root,
    mode => 0644,
    require => Package['cronie'],
}
```

定义一个 array 给 source 的含义是，从第一个开始尝试，应用第一个生效的 source，由于使用了变量，可以做很多有意义的事情。比如：

- ❑ 在 modules/motd/files 里加入 motd_cron.agent1.example.com，以特殊化 agent1 的 source 文件。
- ❑ 如果 host 分级实在太细致了，可以根据一个变量来分组，在 modules/motd/files 里加入 motd_cron.atop，以特殊化 \$dstat_pkg 设置为 atop 的 hosts。
- ❑ 如果有些 agent 在以上 2 个条件的 file 中都找不到，那么使用默认的 motd_cron。

(8) recurse

recurse 表示递归地执行，一般是用在一个文件目录下的属性。下面代码中的 file attribute 都是和 recurse 连用比较有意义的，后面慢慢展开。

(9) recurselimit

recurselimit 指应用到第几层目录，1 就是该目录下，2 就是 2 层目录，和 recurse 连用。

(10) ignore

ignore 指忽略哪些文件，可以使用正则和 recurse 连用。比如恼人的 vim 临时文件，总

会有人编辑到一半，把这种文件推到服务器上，这时就可以用下面的例子。

```
file { '/usr/local/bin':
  recurse => true,
  ignore => "*.swp"
  owner => "root",
  group => "root",
  mode => "0755",
}
```

注意，这里使用的是 Ruby 通配符，与 shell 有点类似，像 `*` 这样的正统 regex 是不接受的。可以接受 `?`、`[]`、`{}`。有兴趣的读者可以用 google 搜索 ruby glob 来看具体区别。

(11) purge

purge 指清空，一般和 recurse 以及 force 连用。

(12) force

force 包含如下两种情况：

- ☐ 子文件夹也一并清空，和 recurse 及 purge 连用。
- ☐ 替换任何子文件夹为源文件类型，甚至会替换成普通文件，和 recurse 连用。

(13) links

links 指在递归管理时，碰到 links 该如何处理，和 recurse 连用意义比较大。

- ☐ follow 会 copy link 的真实文件。
- ☐ manage 会 copy link 自己。
- ☐ ignore 会直接无视。

3. service 配置管理

结束了 file resource 管理，相信读者已经入门，接下来趁热打铁，引出 resource 管理的另一重头戏 service。毕竟虽然 Linux 上一切皆文件，文件从 package 来，但是真正对用户有意义的是，提供一个可靠 service！

熟悉 Linux 的读者肯定曾有过这样的经历，这个程序怎么没有 reload？这个程序怎么没有 graceful restart？这个程序来自于开源项目 init 的命名有些特立独行，又或者这个程序是本公司 Java 攻城狮开发的，什么都没有……面对 Linux 如此精彩的 service 管理案例，下面通过三种情况来说明 service 的相关 attribute 的使用场景，当然本书是基于 CentOS 环境来展开的。

(1) 标准的 CentOS 服务

特点：

- ☐ 有 init 脚本。
- ☐ init 脚本有 status 参数。
- ☐ init 脚本有 graceful restart 或者 reload 参数。
- ☐ init 脚本和 binary 命名符合规范。

相关 attribute 如下：

- ❑ ensure：确定 service 的状态。一般为 running，极个别的情况会使用 stopped。
- ❑ enable：开启自启，true 或者 false。
- ❑ hasrestart：是否有 restart 命令，默认是 false，Puppet 会运行 stop、start 命令。
- ❑ restart：指定 restart 的命令，如果 hasrestart 设置为 true，Puppet 默认会使用 init 的 restart 参数。由于有些优秀的 service 还提供了 reload 功能，可以在线更改配置，如 Nginx 和 Haproxy，因此没必要使用 restart，例如 restart => '/etc/init.d/nginx reload'。

(2) 非标准的 Centos 服务

特点：

- ❑ 有 init 脚本。
- ❑ init 脚本有 status 参数。
- ❑ init 脚本没有 graceful restart 和 reload。这是硬伤，推荐采用搭多个节点的方式来实现 HA。当然如果能承受几秒内的 service 不可达，那么也行。
- ❑ init 脚本或 binary 不符合命名符合规范。

相关 attribute 只有一个，如下：

- ❑ name：表示 init 脚本名字。开源软件最常见的问题是，明明 binary 是 ServiceA，init 脚本为 ServiceAd 或 init 脚本完全是另外一个名字，比如 ServiceAMaster，而 init 脚本的名字就只剩 ServiceA，没有了 Master 这个后缀。笔者碰到的一个问题就是 supervisor 在 CentOS 5 里面是 supervisor，在 CentOS 6 里面是 supervisord，为了解决这个问题，用 name 结合 if 语句可以写出比较整洁的代码，如下：

```
$supervisor_name = $operatingsystemrelease ? {
    /^6/ => "supervisord",
    default => "supervisor"
}

service { "supervisord":
    name => $supervisor_name,
    ensure => running,
    enable => true,
    hasrestart => true,
    hasstatus => true,
    require => [File['/etc/supervisord.conf'], Package["supervisor"]],
}
```

(3) 啥都没有的服务 (Java 程序)

特点：

- ❑ Java 工程师会给你一个命令行去 start 一个 Java 程序。
- ❑ Java 工程师告诉你可以用 kill -9 命令去 stop 一个 Java 程序。

- ❑ Java 工程师会告诉你访问网页去看一个 Java 程序状态。
- ❑ Java 工程师会告诉你他 fork 出来的子进程，可以靠 `grep` 一个关键字获得该 Java 程序的所有进程。

看到这里，读者心中肯定已经万马奔腾，以下 `attribute` 笔者不想说得太细。因为作为一个优秀的 SA，应当做出合理的 `init` 脚本来填补这一神坑。当然还可以提出命令行检查 `status` 的需求。

- ❑ `binary`：指定 `binary` 路径。
- ❑ `start`：指定 `start` 命令。
- ❑ `stop`：指定 `stop` 命令。
- ❑ `hasstatus`：脚本是否有 `status` 参数，`hasstatus` 默认为 `false`，设为 `true` 时，不用设置 `status`。
- ❑ `status`：指定 `status` 命令，如果设置这个值，`hasstatus` 需设为 `false`。
- ❑ `pattern`：如果懒得设置 `hasstatus` 和 `status`，告诉 Puppet `grep` 什么关键字可以查到 `status`。

示例代码如下：

```
$supervisor_name = $operatingsystemrelease ? {
  /^6/ => "supervisord",
  default => "supervisor"
}

service { "supervisord":
  name => $supervisor_name,
  ensure => running,
  enable => true,
  hasrestart => true,
  hasstatus => true,
  require => [File['/etc/supervisord.conf'], Package["supervisor"]],
}
```

4. exec 配置管理

如果说 `package`、`file` 和 `service` 是 `resource` 管理的三板斧的话，`exec` 就是玄铁重剑，虽然威力无穷，但是驾驭起来不易。照 Puppet 官方说法是，如果目前的趋势是官方已有的 `resource` 类型无法满足你的需求，那么可以用一系列 `exec resources` 来管理应用，但如果用了很复杂 `exec` 来管理应用，那就得认真仔细考虑清楚为什么要这么做，是否可以抽象出来并使用自定义的 `resources`（自定义 `resources` 我们会在后文介绍），来让你的 Puppet 更易于维护。

要开始用 `exec` 前，先强调以下 2 点。

1) `exec resource` 最好做到幂等性。

何为幂等性？也就是多次运行对系统产生的影响是相等的，不会对系统产生不好的影

响，除非使用后文会介绍的 `refresh/refreshonly/onlyif/unless/creates` 来严格控制运行条件。

2) `exec resource` 运行条件测试很重要。

根据笔者以往的经验来看，写 `exec` 要执行的 `command` 一点也不难，难的是定义运行条件，和创造测试条件并测试，所以建议在写 `exec` 的时候先写运行条件和测试计划，就好比好的开发是先写测试再写代码的情形一样。

接下来分为三部分来讲解 `exec` 的 `attribute`。

(1) 运行的方式管理

- ❑ `command`：要执行的完整命令行，默认为 `title` 名字。建议 `title` 名字用于描述该 `exec` 资源的作用，而单独使用 `command` 来指定要执行的完整命令。
- ❑ `cwd`：运行时的目录。
- ❑ `environment`：运行时的额外环境变量。Linux 的 `env`，如果在这里面设置 `path` 变量，会覆盖该 `resource` 的 `path attribute`。多个 `environment` 用 `array` 来指定。
- ❑ `user`：运行时的用户，默认 `root`。
- ❑ `group`：运行时的组，默认 `root`。
- ❑ `path`：运行时的 `path` 变量。

学到这里，应该可以简单地写出如下代码了：

```
exec { 'how to run':
  user => 'puppet',
  group => 'puppet',
  cwd => '/tmp',
  command => '/usr/bin/id > /tmp/how_to_run; pwd >> /tmp/how_to_run; /usr/bin/
  env >> /tmp/how_to_run'
}
```

由于该 `exec resource` 是幂等的，对系统不会有任何不好的影响，因此可以放心地让其多次执行。

(2) 运行的结果管理

- ❑ `umask`：Linux 的 `umask`，控制 `exec` 产生文件的 `mode`。
- ❑ `returns`：期待的命令返回值，默认是 0，如果与期待的不符，报错，或者依照 `tries attribute`，多次执行。
- ❑ `tries`：因为返回值不符或者 `timeout`，总共 `try` 的次数，默认为 1，即不 `retry`。
- ❑ `try_sleep`：`try` 之间的间隔时间。
- ❑ `timeout`：一次 `try` 的 `timeout`，默认为 300 秒。
- ❑ `logoutput`：何时输出日志，默认为 `on_failure`，可以选 `true` 和 `false`。

学到这里，即可优化上述代码：

```
exec { 'how to run':
  user => 'puppet',
```

```

group => 'puppet',
cwd => '/tmp',
command => '/usr/bin/id > /tmp/how_to_run; pwd >> /tmp/how_to_run; /usr/bin/
env >> /tmp/how_to_run
tries => '3',
try_sleep => '5',
timeout => '10',
}

```

(3) 运行的先决条件管理

- ❑ **creates** 指如果一个文件不存在，那么运行。这个参数完全可以用 **unless** 和 **onlyif** 替换，只要熟练使用 Linux 的 **test** 命令。示例如下：

```

exec { 'how to run':
  command => '/usr/bin/id > /tmp/how_to_run; pwd >> /tmp/how_to_run; /usr/bin/
  env >> /tmp/how_to_run
  creates: "/tmp/how_to_run",
}

```

- ❑ **onlyif** 表示如果满足该条件，那么运行。满足的定义是该命令返回值为 0。示例如下：

```

exec { 'how to run':
  command => '/usr/bin/id > /tmp/how_to_run; pwd >> /tmp/how_to_run; /usr/bin/
  env >> /tmp/how_to_run
  onlyif: "test ! -f /tmp/how_to_run",
}

```

- ❑ **unless** 表示如果不满足该条件，那么运行。不满足的定义是该命令返回值不为 0。示例如下：


```

exec { 'how to run':
  command => '/usr/bin/id > /tmp/how_to_run; pwd >> /tmp/how_to_run; /usr/bin/
  env >> /tmp/how_to_run
  unless: "test -f /tmp/how_to_run",
}

```

细心的读者可以发现，上述这些控制条件都是希望在特定的情况下才执行 **exec resource** 的操作，因为 **Puppet agent** 默认间隔 30 分钟跑一次，如果真有每 30 分钟操作一次的需求，那么应该使用系统的 **cron** 任务更时适合。

除了上述这些控制条件外，另外还有两个控制条件“**refreshonly**”和“**refresh**”，这两个的使用场景是在 **exec resource** 收到 **refresh event** 的时候控制的，在 **Puppet** 的 **resource** 中，**refresh event** 是一个非常重要的控制 **resources** 之间关系的一个机制。与 **refresh event** 相关的 2 个参数是 **notify** 和 **subscribe**，也就是通知和订阅的意思，下面用一个简单例子来说明一下。

 **注意** 这2个参数 `notify` 和 `subscribe` 又叫 `metaparameter` (元参数), `metaparameter` 的具体内容后面讲解, 目前所要知道的是, 该参数可以在任何 `resource` 中当 `attribute` 用。

```
file { ['/etc/ssh/sshd.conf']:
  source: "puppet:///modules/ssh/sshd.conf",
  validate_cmd => '/usr/sbin/sshd -t -f %',
  notify => Service['sshd'],
}

service { 'sshd':
  ensure => "running",
}
```

当 `file resource` `'/etc/ssh/sshd.conf'` 发生改变时, 会 `notify Service['sshd']`, 即发送一个 `refresh event` 的通知给 `Service sshd`, 注意在使用 `notify` 时, 格式为 `resource type` 首字母大写和一个 `array` 的方式来引用我们要 `notify` 的 `resource`。而 `service resource` 收到这个 `refresh event` 时 (即被 `notify` 时), 默认动作是 `restart` 该 `service`。因此这个示例完成了一个典型的 `service` 维护操作, 即发现有配置文件更新时, 自动重启该 `service`。

现在, 换一种方式实现:

```
file { ['/etc/ssh/sshd.conf']:
  source: "puppet:///modules/ssh/sshd.conf",
  validate_cmd => '/usr/sbin/sshd -t -f %',
}

service { 'sshd':
  subscribe => File['/etc/ssh/sshd.conf'],
  ensure => "running",
}
```

这里的 `service resource 'sshd'` 会主动订阅 `File['/etc/ssh/sshd.conf']`, 当 `File['/etc/ssh/sshd.conf']` 发生改变时, `service resource 'sshd'` 会触发 `restart` 动作。同样, `subscribe` 时, 使用 `resource type` 首字母大写和一个 `array` 的方式来引用 `subscribe` 的 `resource`。

至此, 关于 `refresh event` 和 `subscribe/notify` 的原理及用法, 已经介绍完毕, 最后, 来介绍上文提到的 `exec resource` 最后两个控制条件 “`refresh`” 和 “`refreshonly`”。

□ `refreshonly` 表示只有在收到 `refresh event` 时, 且

- 该 `exec resource` 订阅 (`subscribe`) 的 `resource` 有变更的情况时
- 或者其他 `resource` 主动通知 (`notify`) `exec resource` 时

才会执行该 `exec resource`。示例如下:

```
file { ['/etc/nginx/nginx.conf']:
  source: "puppet:///modules/nginx/nginx.conf",
}
```

```
exec { 'Nginx Reload':
  command => "/etc/init.d/nginx reload",
  subscribe => File['/etc/nginx/nginx.conf'],
  refreshonly => true,
}
```

上文在介绍 service 时，用 nginx reload 来替换 service 的 restart attribute，这次保留了 service nginx 的原始 restart，用 exec 来单独定义 reload。refreshonly => true，保证了只有在 subscribe 的 File['/etc/nginx/nginx.conf'] 改变后，才执行 reload 命令（如上文中提到的，exec 如果不加入控制条件的话，每 30 分钟跑 Puppet agent 的时候都会运行，对于 Nginx reload 来说，如果不用 refreshonly 控制，显然不合适）。

- ❑ refresh 表示在接受到 refresh event 时，指定另外一个命令执行，否则运行两次 exec resource 中的命令。（如上文中提到，这相当于不控制 exec resource，每 30 分钟跑 puppet agent 的时候都会运行，不如用一个系统的 cron 更适合，因此，这里不再对这一个畸形的控制条件展开说明。）

5. 其他资源管理

之前本没有打算写这一部分内容，但是想分享笔者的见解，也算是对读者继续深入其他 resource 的一些建议。

Puppet 目前对于内置 resource 的扩展不是特别热衷，从 Puppet 2.x 一直到 Puppet 4.x，内置 resource type 基本还是那些，即上文提过的 package、file 和 service 三板斧加上 exec 玄铁重剑，使用它们基本可以应付一切 Linux 资源管理。更复杂的场景可以通过组合这四种内置 resource 来自定义 resource，因此对于其他已有的 resource，笔者建议奉行 Linux 一切皆文件的方式来管理。

具体分析如下：

- ❑ 对于配置文件类，比如 interface/cron/mount/vlan/router/host/yumrepo，熟悉 Linux 的读者肯定已经有相关配置文件的经验，没必要重新学习 Puppet 语法，而不是用配置文件的方式来管理这些资源。
- ❑ 对于用户类，比如 user/group/sshkey/ssh_authorized_key，强烈建议搭建 ldap 来解决。理由很简单，对于成百上千机器和种类繁多的 Linux 运维工具，集中化管理认证是相当重要的，ldap 作为各大开源软件默认的认证集中化系统，搭建它真的物有所值，你绝对不会后悔花那么多时间去初始化它。因为后期维护成本几乎为 0，省下的各个系统的认证管理成本是非常可观的。

9.3.2 深入 metaparameter

前文提到了 notify 和 subscribe，并且称它们为 metaparameter（元参数），从字面上理解就是 resource 最基本的参数，可以作为任何 resource 的 attribute，即便是以后会讲到的自定

义的 resource，也可以使用 metaparameter。当然元参数的功能多种多样，接下来将根据功能分类来一一详解。

1. resources 之间的运行控制

(1) require/before

require/before 用于顺序控制，示例如下：

```
package { 'cronie':
    ensure => installed,
}

service { 'cron':
    name      => 'crond',

    ensure    => running,
    require   => Package['cronie'],
}
```

上述示例在 service 'cron' 中定义了 require Package['cronie']，即在 service 'cron' 资源执行之前，需要先安装 package 'cronie'。

逆向思维爱好者可以在资源 package 'cronie' 中使用 before Service['cron']，也能达到同样效果，但建议一个项目使用同一种风格，下文中的 notify/subscribe 也是同理。

(2) notify/subscribe

notify/subscribe 属于 refresh 机制。前文提过，refresh 机制是指针对 resource A 更改后，会触发一个事件给 resource B。对于这些概念，说明如下：

- ❑ resource A：一般指的是 file resource。
- ❑ resource B：一般指的是 service 和 exec resource，其中 service refresh 就是触发 restart 动作；exec refresh 就是再跑一遍该条命令，或者是在 exec 中特别指定了 refresh 的 attribute 作为刷新时要跑的命令。
- ❑ notify：相当于隐式地定义了 before 的顺序关系。
- ❑ subscribe：相当于隐式地定义了 require 的顺序关系。

2. resource 运行控制

(1) noop

noop 表示是否把该 resource 在 noop 的模式中运行，默认 false。noop 的全称是 no operation，它的意思一目了然，就是不进行任何操作，只是测试。不过，建议直接用命令行全局 noop 运行一次，直观又不用改 Puppet 代码。示例代码如下：

```
[root@agent1 ~]# puppet agent -t --noop
.
.
Notice: /Stage[main]/Motd/File[/tmp/ds/1]/ensure: current_value directory, should
```



```
be absent (noop)
Notice: Finished catalog run in 0.76 seconds
```

(2) loglevel

loglevel 是日志级别。默认是 notice，可以设置为 debug、info、notice、warning、err、alert、emerg、crit，同样，通常需求是要看更多的日志来调试的，建议直接用命令行来查看，如下：

```
[root@agent1 /]# puppet agent -t --debug
```

(3) schedule

schedule 即调度时间。说白了就是以 cronjob 方式定期执行 Puppet 某个 resource。但笔者并不推崇，因为 Linux 管理员大多数还是用标准 cron 来统一管理定期比较透明，易于维护。示例如下：

```
schedule { 'everyday':
  period => daily,
  range  => "2-4"
}

exec { ["/usr/bin/apt-get update":
  schedule => 'everyday'
]
```

3. resource 的元属性

(1) alias

alias 是 resource 的别名。给 resource 起别名，可以方便其他 resource 用 metaparameter 引用它。笔者觉得更好的习惯是：

- ❑ 使用 file/service 中的 name attribute 定义易用的 title 名字。
- ❑ 使用 exec 中的 command attribute 定义易用的 title 名字。

(2) audit

audit 用于审计 resource 的改变。审计指定的 resource 的 attribute，即当这个 resource 的 attribute 发生变化时，会在日志中进行记录（日志格式都是以 [audit] 作为前缀），用处中规中矩，后文介绍的监控 Puppet 的工具可以完美替代它。示例代码如下：

```
file { ['/etc/hosts':
  owner => "root",
  group => "root",
  mode  => "0644",
  source => "puppet:///modules/network/hosts",
}]

file { ['/etc/hosts':
  audit => [ owner, content ],
}]
```

(3) tag

tag 用于给 resource 打 tag。tag 包含很多的内容，这里会花时间介绍下基本概念和简单应用，高级应用需要在学习其他高级概念后才可以得心应手。

tag 是一个可以用来分类的属性。它包括如下两种生成方式。

第一种是自动生成。针对一个 resource，它的 tag 自动产生的内容是：

- ❑ resource 的 type，通常是 file、package、service、exec 等。
- ❑ resource 的 title 名字。
- ❑ resource 的上层 container 的类别和 title 名字，通常是 class。
- ❑ 继承上层 container 的 tag。

从 master 上获得一个 agent，它所有 resource 所包含的 tag 如下：

```
[root@puppet /]# puppet master --compile agent1.example.com | grep -E
  "title"|"tags" | tail -10
  "tags": ["class", "cronie", "node", "motd", "default", "package"],
  "title": "cronie"
  "tags": ["epel-release", "class", "node", "motd", "default", "package"],
  "title": "epel-release"
  "tags": ["class", "node", "motd", "default", "package", "yum-plugin-downloadonly"],
  "title": "yum-plugin-downloadonly"
  "tags": ["class", "node", "motd", "default", "package", "apache"],
  "title": "apache"
  "tags": ["service", "class", "cron", "node", "motd", "default"],
  "title": "cron"
```

第二种是手工添加。可以用以下方式给 resource 手工添加 tag。

```
class motd {
  tag(['hello1', 'hello2'])

  file { ['/usr/local/bin/generate_motd.sh':
    source => "puppet:///modules/motd/generate_motd.sh",
    owner => root,
    group => root,
    mode => 0775,
    tag => "script_motd",
  ]

  file { ['/etc/cron.d/motd_cron':
    content => template("motd/motd_cron.erb"),
    owner => root,
    group => root,
    mode => 0644,
    require => Package['cronie'],
  ]
}
```

在上述代码中，用 `tag()` function 给 class `motd` 中所有的 resource 都加上了 `hello1` 和 `hello2` 的 tag。此外，还用 tag metaparameter 给 file `'/usr/local/bin/generate_motd.sh'` 加了一个 tag 的 attribute，为 `"script_motd"`。

那么 tag 有哪些用途呢？

第一，它可限制 Puppet agent 应用于哪些 resources。比如，从 agent 上指定要应用带有 `"script_motd"` tag 的 resources，命令如下：

```
[root@agent1 /]# puppet agent -t --tags script_motd
```

这样一来，Puppet 这次运行就只会复制 file resource `'/usr/local/bin/generate_motd.sh'` 了。

第二，可用于搜索出相应的 resource collector（资源收集器）。

简单用一个例子说明，先看以下第一眼看上去像天书一样的 Puppet 代码：

```
file { ['/etc/httpd/conf/http.conf':
  ensure => file,
  owner  => apache,
  group  => apache,
  mode   => 0644,
  source => "puppet:///modules/apache/http.conf",
  tag    => "apache_config",
}]

file { ['/etc/httpd/conf.d/sitel.conf':
  ensure => file,
  owner  => apache,
  group  => apache,
  mode   => 0644,
  source => "puppet:///modules/apache/sitel.conf",
  tag    => "apache_config",
}]
```

```
Package['httpd'] -> File <| tag == 'apache_config' |>
```

这里重点讲解一下代码中的 `Package['httpd'] -> File <| tag == 'apache_config' |>`。

- ❑ `<|` 和 `|>` 包裹的是一条 search 语句，搜索的条件为有 `apache_config` 的 tag，然后结果集就成为了一个 resource collector，类型是 File。
- ❑ `->` 是 before metaparameter 的另外一种表现形式，意思是 Package 的这个 resource 要在 File 的 resource 之前运行。

4. class 的运行顺序控制

stage 代表 puppet 运行阶段。这个 stage 是一个特殊的 metaparameter，因为它只能用在 class 这一级，是用来控制多个 class 之间的运行顺序的。看一下相关的示例。

site.pp 的示例代码如下：

```
stage { 'pre':
```

```

    before => Stage['main'],
  }

  stage { 'post':
    require => Stage['main'],
  }

```

modules/yum-update/manifests/init.pp 的示例代码如下：

```

class { 'yum-update':
  stage => 'pre',
}

```

对于上述代码，说明如下：

- ❑ stage 的定义是在 site.pp 中，因为它是控制所有 class 的运行顺序的。
- ❑ 默认的 stage 是 main，所以这里定义了 2 个新的 stage，即 pre 和 post，并定义了顺序。
- ❑ 在 class yum-update 中定义了它是运行在 stage pre 中的，也就是在所有 class 之前运行。

9.3.3 深入 fact

fact，在英文中是一个很常用的单词，老外都喜欢用这个单词来描述一个事物或人所具有的实际情况。比如 I have three facts, tall, rich, handsome，翻译成中文就是：我有三个特征：高富帅。好了，言归正传，这节要讲的事实，是 Puppet 用来描述一个机器 fact（实际情况）的工具。fact 可以是这台机器的硬件情况，也可以是这台机器的系统情况、软件情况，甚至是用户自定义的情况。想要看一台 agent 上所具有的事实（实际情况），可以用 facter 这个命令查看，运行不加参数会得到默认 fact，如果加上 -p 参数，则可以得到 Puppet agent 在运行的时候，Puppet master 给予该 Puppet agent 的 fact，比如用户在 Puppet master 模块中自定义的 fact，这些 fact 都可以在 Puppet 的代码中运用，示例如下：

```

[root@agent1 /]# facter | grep -i centos
operatingsystem => CentOS

```

```

[root@agent1 /]# vim manifests/init.pp
file { '/tmp/dsl':
  ensure => file,
  owner  => root,
  group  => root,
  mode   => 0644,
  content => "$operatingsystem"
}

```

```

[root@agent1 /]# cat /tmp/dsl
CentOS

```

初看 fact 貌似没什么用，接下来会根据不同大类列出常用的 fact 和一些使用场景。

1. hardware 类的 fact

❑ blockdevices => sda 为块设备名字。

❑ blockdevice_sda_model => ServeRAID M5110 为块设备类型，有的 RAID 卡会显示 RAID 型号。

❑ blockdevice_sda_vendor => IBM 为块设备的供应商。

来看个示例，根据不同的 RAID 卡型号选择不同的安装包，代码如下：

```
if $blockdevice_sda_model =~ /.*/RAID.*/ and $blockdevice_sda_vendor == "IBM" {
    package {"raidman":
        ensure => installed,
    }
}
```

这里用到 if 的条件语句，用 {} 来包含一个 resource。关于流程控制，会在下文中详解。=~ 跟上 /regex/, 是 Puppet 中的正则匹配的格式。and 表示要同时满足左右 2 个条件；满足 IBM RAID 的，安装 raidman 这个官方 raid 卡管理的 rpm 包。

❑ processorcount => 24 和 physicalprocessorcount => 2 表示处理器的数量和核数。

下面的示例会根据不同的核数来定义配置 php-fpm.conf。

```
[root@agent1 /]# vim manifests/templates/php-fpm.conf
pm.max_children = <%= @processorcount.to_i * 2 -%>
pm = static
```

php-fpm 是笔者非常喜欢的 php fastcgi 进程管理器，它可以有 slowlog、简单的 debug 日志、Apache 风格的进程数量管理，而且支持 reload，优化内存，简直就是 PHP 运维人员的福音。

max_children 配合 static 模式是一种常见的运行模式，至于为什么要设置成和内核数的 2 倍，纯属笔者个人经验和喜好，这里不再展开。

2. kernel os 类的 fact

❑ architecture => x86_64

❑ kernelrelease => 2.6.32-431.el6.x86_64

❑ operatingsystem => CentOS

❑ operatingsystemrelease => 6.6

❑ osfamily => RedHat

❑ is_virtual => true

❑ virtual => docker

上述 fact，相信大家都看得懂，这里就不再赘述。提一点，笔者的虚拟环境是 Docker，有兴趣的读者可以自行研究，绝对是做实验和配置开发环境的利器。

关于使用场景，对于管理了多种 Linux 发行版的读者，这些 kernel os 类的 fact，可以灵活运用于流程控制语句。虚拟机的这 2 个参数，发挥的场景较多，下面给出一个例子，使用 tuned 来调优 kvm guest 机，manifests/init.pp 中的代码如下。

```
if $virtual == "kvm" {
    package {"tuned":
        ensure => installed,
    }

    exec {"tune for kvm guest":
        command => "/usr/sbin/tuned-adm profile virtual-guest",
        unless => "/usr/sbin/tuned-adm active | grep -q virtual-guest",
        require => Package["tuned"],
    }
}
```

tuned 工具的用途和使用方法，可以参阅 redhat 官方文档，地址为 https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html-single/Virtualization_Tuning_and_Optimization_Guide/index.html#chap-Virtualization_Tuning_Optimization_Guide-tuned。

exec "tune for kvm guest" 只有在 "/usr/sbin/tuned-adm active | grep -q virtual-guest" 返回 True 的情况下才不会运行 command/usr/sbin/tuned-adm profile virtual-guest。

3. system 配置类的 fact

- ❑ domain => example.com
- ❑ fqdn => agent1.example.com
- ❑ hostname => agent1
- ❑ interfaces => eth0,lo
- ❑ ipaddress => 172.17.0.22
- ❑ ipaddress_eth0 => 172.17.0.22
- ❑ ipaddress_lo => 127.0.0.1
- ❑ macaddress => 02:42:AC:11:00:16
- ❑ macaddress_eth0 => 02:42:AC:11:00:16
- ❑ netmask => 255.255.0.0
- ❑ netmask_eth0 => 255.255.0.0
- ❑ netmask_lo => 255.0.0.0
- ❑ network_eth0 => 172.17.0.0
- ❑ network_lo => 127.0.0.0

以上 facter 都容易理解，这里不再赘述。下面来讲使用场景，定义 sshd 的监听 ip 的示例代码如下：

```
sshd/templates/sshd.conf
```

```
ListenAddress <%= @ipaddress_eth0 -%>
```

上述代码的目的是为了监听 ssh 在内网中的地址，默认 ssh 监听的是 0.0.0.0。

为什么一定要用 ipaddress_eth 这个 fact 呢？因为 sshd 不支持以 interface 作为监听地址（是的，比较无奈）。

4. 动态系统资源类的 fact

- ❑ memoryfree => 26.68 GB
- ❑ memorysize => 31.20 GB
- ❑ partitions => {"sda1"=>{"size"=>"204800"}, "sda2"=>{"size"=>"1929379840", "mount"=>"/etc/resolv.conf"}, "sda3"=>{"size"=>"2097152"}}
- ❑ swapfree => 1016.34 MB
- ❑ swapsize => 1023.99 MB

目前笔者暂未使用这些 fact，因为笔者坚信好的监控系统才是关键，通过系统负载情况自动做一些 magic 的事情总归是不靠谱的。比如自动调整配置文件的内存使用，总会担心自己写的应变策略会不会被一种极端情况引起不可预期的情况，墨菲定律告诉我们，好的系统管理员还是要及时响应异常，仔细分析，才能使问题迎刃而解。

9.3.4 深入流程控制

1. 条件语句

(1) if 语句

if 语句可以和 elsif、else 连用，判断句可以是如下形式。

第一种：Variables（变量）。

变量可以是如下种类。

- ❑ Strings：空值是 false，非空值是 true。如果值是 "false"，判断下来也会是 true。如果要更加智能地解析是 true 还是 false，可以使用 puppet stdlib(standard library) 中的 str2bool function，它可以把 string 为 '1'、't'、'y' 和 'yes' 解析成布尔型的 true；把 string 为 '0'、'f'、'n' 和 'no' 解析成布尔的 false。具体如何安装 stdlib 模块，会在后文详解。
- ❑ Numbers：任何数字都是 true。类似的，如果要智能解析 0 为 true，1 为 false，要用 stdlib 中的 num2bool function。
- ❑ Undef：任何没有定义的变量都为 false。
- ❑ Arrays and Hashes：任何 array（数组，相当于 Python 中的 list）和 hash（哈希，相当于 Python 中的 dict）都为 true，即便是空 array 或者空 hash。

第二种：Expressions（表达式，下节详解）。

第三种：Functions that return values（一个 function 的返回值，后文详解 function）。

function 的返回值可以是 true 和 false 这样的布尔型，也可以是任何其他数据类型，具体判断可参考 variable 的判断方法。

来看个示例，guest 虚机不需要 ntp。

modules/ntp/manifests/init.pp 中的代码如下：

```
class ntp {
    file {'/etc/ntp.conf':
        ensure => file,
        content => template('ntp/ntp.conf'),
    }

    package {'ntp':
        ensure => installed,
    }

    service {'ntp':
        ensure => running,
    }

    Package['ntp'] -> File['/etc/ntp.conf'] -> Service['ntp']
}
```

modules/ntp/manifests/disabled.pp 中的代码如下：

```
class ntp::disabled{
    package {'ntp':
        ensure => purge,
    }
}
```

modules/kvm-ntp/manifests/init.pp 中的代码如下：

```
if str2bool($is_virtual) {
    include ntp::disabled
}
else {
    include ntp
}
```

在上述代码中：

- str2bool 是上文提到的 stdlib 中的 function。
- \$is_virtual 是上文提到的内置 facter，value 可为 true 和 false。
- 上文提到默认情况下，即使一个 string 的 variable 写着 false，它也会被 if 当成布尔型的 true，所以可借助 str2bool function 智能解析。
- 在 ntp::disabled 中，:: 是访问不同 namespace 的符号，比如 ntp::disabled 是要访问 modules/ntp/manifests/disabled.pp。
- namespace 即命名空间，如果读者有一点编程知识应该不会陌生，它是用来解决人类词汇量太少而重复命名函数 / 变量 / 类等问题的。直白点说，在 shell 中查看

service A 配置 listen ip 和 service B 配置文件的 listen ip 时, 虽然它们都叫 listen ip, 但是你不会混淆, 因为是在不同路径下的不同文件里, 互相不干扰, namespace 即在编程领域的不同空间下用来存放各种有可能命名相同的 variable/function/class。

(2) unless 语句

和 if 相反, unless 语句没有 elseif 和 else 子句, 其他相同。笔者认为用处不大, 可以用取反号代替。示例代码如下:

```
unless $blockdevice_sda_vendor == "IBM" {
    notify {"say": message => "I am not IBM"}
}

if ! ($blockdevice_sda_vendor == "IBM") {
    notify {"say": message => "I am not IBM"}
}
```

上述两段代码的实现结果一样。! 后面要加上 () 来包含表达式, 由于 ! 永远是优先级最高的, 如果没有 () 来定义顺序, 它会先和 \$blockdevice_sda_vendor 取反, 而这不是我们想要的结果。

(3) case 语句

case 用于对一个变量或有返回值的 function 的各种可能匹配做后续的工作。示例代码如下:

```
case $operatingsystem {
    'Solaris': { include role::solaris } # apply the solaris class
    'RedHat', 'CentOS': { include role::redhat } # apply the redhat class
    /^(Debian|Ubuntu)$/:{ include role::debian } # apply the debian class
    default: { include role::generic } # apply the generic class
}
```

在上述代码中:

- ❑ operatingsystem 是内置 facter。
- ❑ 逗号分隔可以使 2 种匹配合一。
- ❑ // 正斜杠包含的是 regex。
- ❑ default 是一个特殊匹配。相当于 shell case 中的 “*”。
- ❑ {} 包含的是要后续做的工作。

(4) selectors 选择器

就是用 “?” 这种形式的简写来实现不同 variable 的定义工作。笔者觉得正统的 case 比较易于读懂和维护。看两个对比示例。

使用 selectors ? 的示例如下:

```
$rootgroup = $osfamily ? {
    'Solaris' => 'wheel',
    /(Darwin|FreeBSD)/ => 'wheel',
```

```

    default          => 'root',
  }

  file { ['/etc/passwd':
    ensure => file,
    owner  => 'root',
    group  => $rootgroup,
  ]
}

```

使用 case 的示例如下:

```

case $osfamily {
  'Solaris'          => { $rootgroup = 'wheel'}
  /(Darwin|FreeBSD)/ => { $rootgroup = 'wheel'}
  default            => { $rootgroup = 'wheel'}
}

file { ['/etc/passwd':
  ensure => file,
  owner  => 'root',
  group  => $rootgroup,
]
}

```

2. 表达式

(1) 比较运算符

- == 的含义是 equality, 即等于。
- != 的含义是 non-equality, 即不等于。
- < 的含义是 less than, 即小于。
- > 的含义是 greater than, 即大于。
- <= 的含义是 less than or equal to, 即小于或等于。
- >= 的含义是 greater than or equal to, 即大于或等于。
- =~ 的含义是 regex match, 即正则匹配。
- !~ 的含义是 regex non-match, 即正则不匹配。
- in。

对于上面的运算符不做过多解释, 这里 in 有些特殊, 用例子讲解下, 如下:

```

'eat' in 'eaten' # TRUE
'Eat' in 'eaten' # FALSE
'eat' in ['eat', 'ate', 'eating'] # TRUE
'eat' in { 'eat' => 'present tense', 'ate' => 'past tense' } # TRUE
'eat' in { 'present' => 'eat', 'past' => 'ate' } # FALSE

```

in 后面跟 string, 代表了从左开始匹配, 如果前者是后者的子集则为 TRUE; in 后面跟 array, 代表数组中任一元素匹配, 即为 TRUE; in 后面跟 hash, 必须是 hash 的 key 匹配, 才为 TRUE, value 不用作匹配。

(2) 布尔运算符

- ☐ and
- ☐ or
- ☐ !(not)

前文曾提到要注意顺序, 建议多用 () 来显式地说明多个表达式的顺序。

(3) 算术运算符

- ☐ + (加法)
- ☐ - (减法)
- ☐ / (除法)
- ☐ (乘法)
- ☐ % (取模)

有些 string 长得像数字, 这时要用 to_i 的 Ruby 内置 function 来转换数据类型。manifests/templates/php-fpm.comf 中的代码如下:

```
pm.max_children = <%= @processorcount.to_i * 2 -%>
pm = static
```

9.3.5 深入 function

function 在 Puppet 的世界里 2 种类型, 即有返回值的和无返回值的, 官方的称呼是 rvalues 和 statements。另外注意, 所有 function 都是在 master 上执行后, 编译成 catalog 再传给 agent 的。

1. function type: rvalue

(1) content 生成的类型

- ☐ file(): 用于读取文件内容并返回 string。可接受如下参数。
 - 相对路径, 指定 mysql/my.cnf 映射到 modules/mysql/files/my.cnf 路径。
 - 绝对路径, 可以指定任何在 Puppet master 上的文件。
 - 多个参数。和 file resource 里面的 source attribute 类似, 会返回第一个找到的文件, 跳过任何不存在的文件。
- ☐ template(): 用于读取 erb 的 template 文件并返回 string。可以接受的参数和 file 一样, 这里不再赘述, 后文中会详解。
- ☐ inline_template(): 用于内联 template, 也就是不需要 template 文件, 内容直接写在 pp 文件里。

来看个示例, 以下是 motd/manifests/init.pp 中的代码:

```
file { '/etc/cron.d/motd_cron':
  content => inline_template('*/* * * * * root bash /usr/local/bin/generate_
    motd.sh && echo "I installed <%= @apache_pkg -%>" >>/etc/motd\n"),
```

```

owner => root,
group => root,
mode => 0644,
require => Package['cronie'],
}

```

注意，如果内容太长，建议还是用单独的 `template` 文件；命令末尾的 `\n`，对于某些配置文件来说是必须的，比如 `cron`。

(2) content 修整的类型

- ❑ `regsubst()`：代表 puppet 里的 `sed`，在下面的示例代码里，`regsubst` 中的第一个参数是要处理的 `string`，第二个参数是匹配式，第三个是替换式，第四个是标志位，比如 `G` 是 `global` 匹配替换，`I` 是忽略大小写。这个例子是要取 `eth0` ip 的网络段。

```

file { ['/tmp/dsl':
  ensure => file,
  owner => root,
  group => root,
  mode => 0644,
  content => regsubst($ipaddress_eth0, '^(\\d+)\\. (\\d+)\\. (\\d+)\\. (\\d+)$', '\\1\\.\\2\\.\\3'),
}

```

- ❑ `split()`：用于把 `string` 按照匹配的分隔符转换成 `array`。

```

$string      = 'v1.v2:v3.v4'
$array_var1 = split($string, ':')
$array_var2 = split($string, '.')
$array_var3 = split($string, '[:.]')

```

- ❑ `$array_var1` - ['v1.v2', 'v3.v4']，这个数组以 ":" 作为分隔符，得到 2 个元素，'v1.v2' 和 'v3.v4'。
- ❑ `$array_var2` - ['v1', 'v2:v3', 'v4']，这个数组以 "." 作为分隔符，得到 3 个元素，'v1'、'v2:v3' 和 'v4'。
- ❑ `$array_var3` - ['v1', 'v2', 'v3', 'v4']，这个数数以 ":" 或者 "." 作为分隔符，得到 4 个元素，'v1'、'v2'、'v3' 和 'v4'，可以看出 `[:.]` 这个正则，和 `awk -F '[:/]' '{print $1,$NF}' /etc/passwd` 类似。

(3) 判断返回布尔值的类型

- ❑ `defined()`：用于判断一个 `resource type` 或者一个 `class` 是否已经定义，是则为 `true`，否则为 `false`。
- ❑ `tagged()`：用于判断一个 `tag` 是否已定义，是则为 `true`，否则为 `false`。

(4) 其他类型

要说明一下，这里的类型分类是笔者为方便知识梳理自己定义的，类型之间没有特殊区别。下面给出几个有意思的 `rvalue` 型的 `function`。

- ❑ `fqdn_rand()`：随机数。有些时候，想在多台机器上分时段跑一个 `cron`，而不是多台机器同一时刻一起跑（这会对中央系统造成压力，比如远程备份工作），那么可以采用如下代码。在该代码中，`$fqdn_rand(60)` 指定了在范围为 0~60 里为这个 `cron` 取随机数，从而在不同的时间执行该 `cron`。

```
file { '/etc/cron.d/backup_cron':
  content => inline_template("$fqdn_rand(60) * * * * root bash /usr/local/
    bin/backup_transfer.sh\n"),
  owner => root,
  group => root,
  mode => 0644,
}
```

- ❑ `generate()`：从 shell command 中生成内容。它的使用场景很多，这里举个简单的例子，假如 Puppet master 上有一个配置文件用来管理该机器是什么产品，比如是 `product01` 或 `product02`，那么可以采用如下代码。

`/etc/servers_allocation.conf` 中的代码如下：

```
product01: agent1, agent3
product02: agent2, agent4
```

`motd/manifests/init.pp` 中的代码如下：

```
file { ['/tmp/ds1':
  ensure => file,
  owner => root,
  group => root,
  mode => 0644,
  content => generate("/usr/bin/awk", "-F:", "/$hostname/ {print \$1}",
    '/etc/server_allocation.conf')
}

[root@agent1 /]# cat /tmp/ds1
product01
```

2. function type: statement

(1) `contain()`

在 class 依赖顺序中，`contain()` 用来替换 `include`。如果读者是 Puppet 高手，且喜欢 class 之间互相依赖嵌套，那么肯定已经知道 `include` 并不像 `resource` 中的 `before` 和 `require` 一样好用，`before` 和 `require` 可以确保 `resource` 之间的运行顺序，而 `include` 并不能保证 class 之间的运行顺序，它只是简单地告诉 Puppet，在执行 class A 的时候 class B 也要一起执行，所以它们会并行，没法保证 class 中 `resource` 的依赖关系。

于是 `contain` 就横空出现了，可以愉快地用下个例子，指定 class 直接的依赖。

```
class wrapper {
  contain foo
```

```
contain bar
contain end
```

```
Class['foo'] -> Class['bar'] -> Class['end']
}
```

上述代码会严格按照你定义的顺序执行，并且也包括每个 class 中包含的 resource。

(2) include()

include() 是最标准的声明 class 的函数。没有约束 class 之间的执行顺序，项目初期，一般不会有那么复杂的类嵌套和依赖，所以 include 即可。

(3) require()

require() 是 class 间简单依赖。适用于简单的 class 之间的依赖，比如只有一层 class 的依赖，即 class A 依赖于 class B，class B 中没有依赖其他 class。但如果被依赖的 class B，还有另外一层 class C 的依赖，会让代码变得非常难读，不如上文的 contain() function 加上 Class['foo'] -> Class['bar'] -> Class['end'] 那么易懂。而且有些时候 class 之间会有些 require/before/notify/subscribe 的 metaparameter，而 require 不会尊重它们，导致有时 Puppet 出错。因此在处理复杂依赖逻辑的时候，建议还是使用 contain() function。示例代码如下：

```
class bar {
  require foo
  notify { 'bar': }
}
```

```
class foo {
  notify { 'foo': }
}
```

```
include bar
```

(4) realize()

realize() 用于声明 virtual resource。在介绍 realize 之前，先介绍 virtual resource 存在的意义，在很多时候我们想在两个 class 中都定义一个 resource，而且这两个 class 又会被同一个 agent 调用。比如一个 class django 要安装 python-yaml package，另外一个 class nagios 也要装一个 python-yaml 来运行一个 Python 监控脚本，如果在 2 个 class 中都定义了 Package["python-yaml"]，那么会产生重复声明的错误，dirty 的一种方法示例如下。

django/manifests/init.pp 中的代码如下：

```
package {"python-yaml":
  ensure => installed,
}
```

nagios/manifests/init.pp 中的代码如下：

```
package {"python-yaml.x86_64":
```

```

    ensure => installed,
  }

```

可以看到，仅仅是加了一个 x86_64 的后缀，就欺骗了 Puppet，这确实一个很 dirty 的方法。

那么现在看如何用 virtual resource 和 realize() function 完美地解决这个问题。

site.pp 中的代码如下：

```

node default {
    include first_default
}

```

first_default/manifests/init.pp 中的代码如下：

```

@package {"python-yaml":
    ensure => installed,
}

```

nagios/manifests/init.pp 中的代码如下：

```

realize Package("python-yaml")

```

django/manifests/init.pp 中的代码如下：

```

realize Package("python-yaml")

```

上述代码先创造了一个 first_default 的模块，并让所有的 node 都默认加载它（笔者相信这个 default class 是一般项目都需要的，比如控制一些全站都要加的配置 service 和 tools，或者仅控制 include 其他 class）。然后在 first_default 里定义了 virtual resource，@package "python-yaml"，注意 @ 就是定义 virtual resource 的标识。最后，在各个 class 里使用 realize virtual resource 语句。

(5) tag()

tag() 为打标记。上文提过，不赘述。

(6) versioncmp()

versioncmp() 是 version string 的比较 function。这个是一个很有用的小工具，它可以智能解析常用的版本标记方式并比较。versioncmp(\$a,\$b) 会返回如下值：

- 1：表示版本 \$a 高于 \$b。
- 0：表示版本 \$a 等于 \$b。
- -1：表示版本 \$a 低于 \$b。

示例代码如下：

```

if versioncmp('2.6-1', '2.4.5') > 0 {
    notice('2.6-1 is > than 2.4.5')
}

```

看，够智能吧！

(7) debug()、info()、notice()、warning()、err()、alert()、emerg()、crit()

以上为日志 function。也就是在 Puppet 日志中加入相应日志 level 的日志。

(8) fail()

fail() 用于主动抛出个 puppet fail，一般用得较少。

9.3.6 深入 template

前文已提到过 template，当然只是一笔带过，现在来详细分析下它。所谓 template 就是可以当作模版使用，可供 file resource 引用的特殊文件，里面可以用嵌入大量的变量，甚至 ruby 代码片段，也就是说任何会用到第二次的且仅需改动几行的文件，都应该使用 template 来管理它！

当然 Puppet 是基于 Ruby 的，这个 template 也是基于 Ruby 的 erb template 演变过来的，所以对于 template 文件，约定俗成都以 .erb 结尾。template(“my_module/mytemplate.erb”)会映射到 /etc/puppet/modules/my_module/templates/mytemplate.erb。接下来会通过如下几个方面深入了解 template。

1. template 语法

其实，template 语法就是纯文本加内嵌 Ruby 的语法。即便只是一个 @operatingsystem，也是 Ruby 风格 instance level 的变量引用。instance 就是 class 的一个实例，instance level 的变量也就是一个 Puppet agent 拥有的变量，虽然 template 可以使用 Ruby 语法对这些变量做各种变换使用，但是 Puppet 的原则是尽量提供简单的声明式语言，避免大家写复杂的 Ruby 语言，这里也不会把重心放在解释 Ruby 语法上。

- ❑ <%= Ruby 表达式 %>：这个标签最后会被 Ruby 表达式的返回值所替换并嵌入 template 文本内容，最简单的就是用 @ 来引用 instance 变量。
- ❑ <% Ruby code %>：这个标签不会有任何返回值，即不会替换嵌入 template 文本内容，通常是用作变量定义、嵌入条件和迭代等流程控制语句，后文会有详解。
- ❑ <## comment %>：template 的注释，是用户不想传给 agent 的注释，注意如果仅仅是 #，该段文本也会一起被同步到 agent 的这个文件上，通常是 conf 所要的注释。
- ❑ <%-：与 <% 一样，但是会避免加入多余的左空格。在 template 嵌入 Ruby 语法的时候，为了读起来更优雅，难免会多加一个空格和空行，但是有些 conf 文件，并不喜欢这些多余的空格和空行，使用 <%- 可以自动去除。
- ❑ -%>：与上个标签用途一样。

2. template 中的 variable

variable 是 template 的核心，它用来控制每台 agent 上配置的差异，它的来源多种多样，大致可以分为以下几种。

- ❑ `facter`
- ❑ `class` 中定义
- ❑ `site.pp` 的 `node` 中定义
- ❑ `enc` (后文会介绍)
- ❑ `hiera` (`Puppet` 的一个官方工具, `key/value`+ 组织结构化的方式管理每个 `agent` 的差异配置)

以下命令用于查看该 `agent` 所有可用的 `variables`。

```
file { ["/tmp/var.yaml":
  content => inline_template("<% scope.to_hash.each do |k,v| -%><%= 's = %s\n' % [k, v] %><% end -%>"),
}
```

另外, 关于 `variable`, 还有一个常见的情景是要判断外部系统里 (`enc/hiera`) 是否定义了该 `variable`。没有的话, 会用一个默认值。

下面是 `templates/task.erb` 中的代码。

```
I have external task <%- if @task -%>
<%- @task -%>
<%- else -%>
0
<%- end -%>
```

可以看到, 使用了 `<%-` 和 `-%>` 后, 代码看起来更清晰, 虽然看上去分行, 但是结果还是一行字。

`Puppet` 官方比较了 `@variable`、`scope.lookupvar('variable')`、`has_variable? ('variable')` 这三种方式, 推荐使用 `@variable`, 具体原因参考官方文档。

3. `template` 中的迭代

(1) `array` 迭代

下面直接给出示例代码。

`init.pp` 中的代码如下:

```
$values = [val1, val2, otherval]
```

`template` 中的代码如下:

```
<% @values.each do |val| -%>
Some stuff with <%= val %>
<% end -%>
```

`template` 的输出文本内容为:

```
Some stuff with val1
Some stuff with val2
Some stuff with otherval
```

(2) hash 迭代

以下为示例代码：

```
file { "/tmp/var.yaml":
  content => inline_template("<% scope.to_hash.each do |k,v| -%><%= '%s = %s\\n' % [k, v] %><% end -%>"),
}
```

这个例子，是上文提到过的查看该 agent 所有可用的 variables 的示例，在此具体分析。对于 `|k,v|`，由于 hash 有 key 有 value，因此用 2 个变量名来迭代。`'%s = %s\\n' % [k, v]` 的意思是用 `k = v` 的形式表达出来，`%s` 是一个联合变量和文本的一个 tricks。

4. template 中使用 function

这里的 function 是指 rvalue 类型的 function，如 `file()`、`template()`、`generate()`，语法形式为：

- ❑ 以 `function_` 作为前缀来引用 function。
- ❑ 参数必须是一个 array，即使只有一个参数。

下面是示例代码。

`files/default_conf` 中的代码如下：

```
LogLevel = Info
User = root
```

`templates/full_conf.erb` 中的代码如下：

```
ListenIP = <%- @ipaddress_eth0 -%>
<%= scope.function_file(["my_module/default_conf"]) %>
```

这看上去有点画蛇添足，但当配置文件相当长，且需维护多个不同生产的文件时，分段的配置文件更易于管理。

9.3.7 深入 define type

`define type` 是一个在 `pp` 文件里可以定义的函数，可以避免撰写雷同的 Puppet 代码。同样来看示例。

`modules/zabbix-proxy/manifests/init.pp` 中的代码如下：

```
define zabbix_proxy_directory_permission_ensure {
  file { ["${title}":
    ensure => directory,
    owner  => zabbixsrv,
    group  => zabbix,
    mode   => 0775,
    require => Package["zabbix-proxy"];
  ]
}
```

```

zabbix_proxy_directory_permission_ensure {
  [
    "/var/lib/zabbixsrv",
    "/var/run/zabbixsrv",
    "/var/log/zabbixsrv",
  ]:
}

```

这里 define 定义了 zabbix_proxy_directory_permission_ensure 的 type, \${title} 是默认的参数, 不用显式定义, 它代表了传进来参数的标题名, 这里就是 “/var/lib/zabbixsrv” “/var/run/zabbixsrv” “/var/log/zabbixsrv”。该段代码会调用 zabbix_proxy_directory_permission_ensure 的 type, 并且传给它一个列表, 让其执行三遍。

当然也有高级用法, 一起来看。

modules/user-manage/manifests/init.pp 中的代码如下:

```

class user-manage {
  define planfile ($user = $title, $content) {
    file {"/home/${user}/.plan":
      ensure => file,
      content => $content,
      mode   => 0644,
      owner  => $user,
      require => User[$user],
    }
  }

  user {'canglaoshi':
    ensure      => present,
    managehome => true,
    uid         => 519,
  }

  user {'tangnvshen':
    ensure      => present,
    managehome => true,
    uid         => 518,
  }

  planfile {
    'canglaoshi':
      content => "Working on new movie";

    'tangnvshen':
      content => "Working on new film";
  }
}

```

这里定义了 `planfile` 的 `define type`，和前一示例不同，它显式地定义了 `$user = $title`（这样，下面使用的时候，可以使用 `$user` 这样比较有意义的变量名），`$content`。在调用 `planfile` 这个 `define type` 时，输入 `$user` 和 `$content` 这两个变量，功能是为两个用户创建了 `planfile`。注意这里的格式，是以分号“`;`”分隔 2 个 `user` 的，并且是用 `puppet hash` 的格式“`=>`”来定义变量值的。

Puppet 实战

本章承接上一章 Puppet 配置管理来讲解一些实战相关的知识点。本章将通过一些简单明了的例子展开，并结合笔者 5 年多 Puppet 实战的项目经验，让读者真正地在项目中运用 Puppet，而不是简单的局限于写写小模块，做做实验。此外，笔者也在一些例子中分享一下这些年遇到的一些坑，希望能帮助读者在运维过程中尽量避免这些坑，或者提供思路，抛砖引玉，使读者可以悟出针对自身项目的解决之道！

10.1 扩展 Puppet

相信读者通过学习上一章的各种例子，已经跃跃欲试了，那么别犹豫，尽情去自动化你的部署配置流程吧。不过，如果你还没有开始动手写 Puppet 的话，建议先别阅读这一节，因为这一节提到的一般是在大量运用 Puppet 后，会碰到的内置功能无法满足实际需求的情况，这种情况一般只会占到一个项目的 10%~20%，甚至少于 10% 或根本没有，这取决于一个项目的复杂度，对于初学者来说，没有必要花费大量时间去研究 10% 的内容。

另外，对于已经入门的读者来说，建议不要重复发明轮子，能用公有模块使用，本节也会主力讲解如何获取更多的公有模块，如何快速入门，当然 Ruby 达人，且是“轮子大师”的请无视……

10.1.1 自定义模块

本节主要讲的是理念和最佳实践，并非是模块的手把手教学，目的是指导读者学会如何使用公有的良好模块来扩充自己的自动化任务，要知道，在 Puppet module repo 里有

3000 多的模块等你来探索！

1. 模块的目录树

一个简单的目录树结构如下：

<MODULE NAME>

- manifests
- files
- templates
- lib
- facts.d
- tests
- spec

下面用一个 `my_module` 示例来详解目录树。

- ❑ `my_module`：是模块的主目录，默认放在 `/etc/puppet/modules`。
- ❑ `manifests/`：manifests 子目录是放所有 `.pp` 清单文件的。
- ❑ `init.pp`：这是一个模块必须要有的 `pp` 文件，而且必须要有一个 `class`，名字和模块的名字一样，比如这里模块名是 `my_module`，那么定义类的时候，一定要 `class my_module {}`。
- ❑ `install.pp`：一个有关安装工作的 `pp` 文件，完整引用方法为 `my_module::install`。对于 `::`，前文说过和 `shell` 的目录分割符 `/` 类似。除了 `init.pp` 文件以外，其他 `pp` 文件都是 `optional` 的，在公有模块内，分割多个 `pp` 管理一个模块的现象很普遍。大多数情况下，一个 `init.pp` 已经够用，希望成为一个自定义模块大师的读者，可以参考公有模块学习一般的分割习惯。
- ❑ `files/`：包含静态 `file`。
- ❑ `service.conf`：这个文件 `source => URL` 应该为 `puppet:///modules/my_module/service.conf`，同样也适用于 `file('my_module/service.conf')`。
- ❑ `lib/`：包括所有自定义的 `resource/provider/function/facter`。
 - `lib/puppet/type`：自定义 `resource`，比如 `my_module/lib/puppet/type/mysql_user.rb`。
 - `lib/puppet/provider`：自定义 `resource` 所需要的 `provider`，比如 `my_module/lib/puppet/provider/mysql_user/my_module.rb`。
 - `lib/puppet/parser/functions`：自定义 `function`，比如 `my_module/lib/puppet/parser/functions/str_to_mysql_password.rb`。
 - `lib/facter`：自定义 `facter`，比如 `my_module/lib/facter/mysql_db_size.rb` ^①。
- ❑ `facts.d/`：包含所有的 `external facts`，可以完美替代 `lib/facter` 下的 `custom facts`。它

① 如果自己写 `facter`，建议用下文的 `facts.d` 来替换 `facter`。

的好处是可执行任何类型的脚本文件，比如 shell/python/ruby 等，只要放在 facts.d 下，输出格式是下列三种之一即可。

第一种：

```
yaml
---
    key1: val1
    key2: val2
    key3: val3
```

第二种：

```
json
{
    "key1": "val1",
    "key2": "val2",
    "key3": "val3"
```

第三种：

```
txt, key=value
key1=value1
key2=value2
key3=value3
```

- ❑ templates/：包含所有 templates。引用方式为 content => template('my_module/component.erb')。
- ❑ tests/：包含该模块的使用范例，帮助他人理解和复用这个 Puppet 模块，因此包含的文件应该和 manifests 下的文件一一对应，认真撰写该目录是要贡献给开源社区的代码标准之一，Ruby 达人和模块大师可以自行研究。
 - init.pp，该文件是范例的主体，指导别人该如何调用这个模块
 - install.pp，如果模块分割比较细致，有多个 pp 文件，那么也需要有相应的范例，比如 install.pp 应该表述了这个模块 install 部分的范例
- ❑ spec/：Puppet 的 unit test，主要使用开源项目 rspec 的框架所衍生出的 rspec-Puppet，理念就是“Behaviour Driven Development for Ruby.Making TDD Productive and Fun”，翻译成中文就是要实现 TDD（测试驱动开发）和 BDD（行为驱动开发）。这其实也是很多公司在推行的开发模式，可以有效减少 bug。同样，认真撰写该目录主要是要贡献给开源社区的代码标准之一，笔者自认没有达到如此高度，Ruby 达人和模块大师可以自行研究。

2. 模块的撰写流程

第一步：思考模块的用途。

在着手写模块之前，思考模块到底要做什么事情，最基本的是，一个模块不要做多件

事情。这里所说的“多件事情”其定义又是什么呢？举个简单的例子，要装一个 LAMP 里面的 Apache、MySQL、PHP，那么最好有三个模块，分别是 Apache、MySQL、PHP。好处有以下三点。

- ❑ 代码重用。比如一个 MySQL 模块，不仅在安装 LAMP 时可以用，以后如果要部署一个内部的管理工具，也需要依赖 MySQL，那么就可以直接使用 `include mysql` 这样的语句进行调用，不需要重复先前的工作。
- ❑ 业务架构的扩容。一般来说项目实施过程中一定会碰到 all-in-one box 到 one-in-each box 的架构变更，把每个 service 组件分开，有利于后期的维护工作。
- ❑ 简洁。一个上千行的模块，会使后面的维护工作变得举步维艰，而且会使工程师之间各自为战，宁愿浪费时间自己重写，也不愿意读懂先前的代码。

为了达到这个最优实践，可以先从下面三个问题开始思考。

- ❑ 你的模块会实现什么样的功能？
- ❑ 它具体在做哪些工作？
- ❑ 它做的工作是否有部分可以在以后的项目中使用？

如果针对第一个和第二个问题得出的回答是实现了 A 和 B，或者在第三个问题中，明确地找出了日后能被重用的部分，那么就应该考虑分割模块了。Puppet 官方建议，一个具有规模的项目中应该有近 200 个模块，在笔者的游戏项目有 90 多个模块，也算是初具规模了。


第二步：做好 Module 结构规划。

Puppet 官方的最佳实践认为，任何一个 class 都应该只做单一的事情，这看起来会比较复杂，笔者的模块并非严格遵守这一最佳实践，但是，如同前文所说，这里的主要目的是让读者知晓最佳实践是什么，并鼓励大家使用公有的模块。

(1) class 设计

模块的 class 设计奉行一个原则，一个 pp 文件包含一个 class 定义，一个 class 包含了一件事情，比如 Puppetlabs-ntp 模块里面包含了如下内容。

- ❑ `ntp/manifests/init.pp`：默认 pp。
- ❑ `ntp/manifests/config.pp`：负责 ntp config file 的管理。
- ❑ `ntp/manifests/service.pp`：负责 ntp service 的管理。
- ❑ `ntp/manifests/params.pp`：负责配置 ntp 的 Puppet 模块，这是主要要改的配置文件，以实现个性化配置。
- ❑ `ntp/manifests/install.pp`：负责 ntp package 安装。

 说明 获取方式是在 master 上执行“`puppet module install puppetlabs-ntp`”命令，具体模块获取会在后文介绍。

下面来看几个示例。

init.pp 中的代码如下：

```
class ntp (
    $autoupdate      = $ntp::params::autoupdate,
    $config           = $ntp::params::config,
    $config_template  = $ntp::params::config_template,
    $driftfile        = $ntp::params::driftfile,
    $keys_enable      = $ntp::params::keys_enable,
    $keys_file        = $ntp::params::keys_file,
    $keys_controlkey  = $ntp::params::keys_controlkey,
    $keys_requestkey  = $ntp::params::keys_requestkey,
    $keys_trusted     = $ntp::params::keys_trusted,
    $package_ensure   = $ntp::params::package_ensure,
    .
    .
    .
) inherits ntp::params {

    validate_absolute_path($config)
    validate_string($config_template)
    validate_bool($disable_monitor)
    validate_absolute_path($driftfile)
    .
    .
    .
    include ::ntp::install
    include ::ntp::config
    include ::ntp::service

    anchor { 'ntp::begin': } ->
    class { '::ntp::install': } ->
    class { '::ntp::config': } ~>
    class { '::ntp::service': } ->
    anchor { 'ntp::end': }

}
```

可以看出 init.pp 主要做了如下 5 件事：

- 1) 继承 class ntp::params，使里面的所有变量都可访问。
- 2) 定义 local variable，把这些 local variable 变成 class 的 parameter，允许将来有外部来源可以调用 class 的同时更改这些 parameter，比如 enc。
- 3) 赋予上述这些 parameter default 值，值为 ntp::params 里的变量值，如 \$autoupdate = \$ntp::params::autoupdate。
- 4) 用 stdlib 里的 function、validate 等变量。
- 5) 用 anchor 和 -> 定义 class 的依赖关系。



说明

上述示例中提到了 `stdlib` 模块，这是一个大幅扩展了 Puppet function 的标准库（类似 CentOS 中的 `Epel` 源），安装 `stdlib` 的命令为 `puppet module install stdlib`，`anchor` 是一个 `stdlib` 的 function。之前在介绍 function 时讲解过 `contain()`，它也是帮助解决 class 之间依赖的一种方式，`contain` 和 `anchor` 同时存在的理由很简单，`anchor` 是社区力量先出现的解决方案，Puppet 官方听取建议，做了相应 function，只不过实现方式不一样。具体哪个好。真是见仁见智！

上述示例中，使用了 `Anchor` 实现的类依赖，`contain()` 方式如下：

```
contain ::ntp::install
contain ::ntp::config
contain ::ntp::service

Class['::ntp::install'] ->
Class['::ntp::config'] ->
Class['::ntp::service']
```

以下是关于 `module::install` 的示例。

`ntp/manifests/install.pp` 中的代码如下：

```
class ntp::install inherits ntp {

    package { 'ntp':
        ensure => $package_ensure,
        name    => $package_name,
    }

}
```

`class install` 继承了 `class ntp`，因此可以使用 `$package_ensure` 这样的变量，这些变量也是 `ntp` 从 `ntp::params` 继承过来并本地化的，且有了 `default` 值。此外，要说明的是，`class install` 只包含了 `package` 的安装。

下面是关于 `module::config` 的示例。

`ntp/manifests/config.pp` 中的代码如下：

```
class ntp::config inherits ntp {

    if $keys_enable {
        $directory = ntp_dirname($keys_file)
        file { $directory:
            ensure => directory,
            owner   => 0,
            group   => 0,
            mode     => '0755',
            recurse  => true,
        }
    }

}
```

```

    }

    file { $config:
      ensure => file,
      owner  => 0,
      group  => 0,
      mode   => '0644',
      content => template($config_template),
    }
  }
}

```

class config 继承了 class ntp，因此可以使用 \$config_template 这样的变量，这些变量也是 ntp 从 ntp::params 继承过来并本地化的，且有了 default 值。class config 只包含了 ntp 的配置文件管理，ntp_dirname 是一个自定义 function，路径为 ntp/lib/puppet/parser/functions/ntp_dirname.rb。

下面是关于 module::service 的示例。

ntp/manifests/service.pp 中的代码如下：

```

class ntp::service inherits ntp {

  if ! ($service_ensure in [ 'running', 'stopped' ]) {
    fail('service_ensure parameter must be running or stopped')
  }

  if $service_manage == true {
    service { 'ntp':
      ensure    => $service_ensure,
      enable    => $service_enable,
      name      => $service_name,
      hasstatus => true,
      hasrestart => true,
    }
  }
}

```

class service 继承了 class ntp，因此可以使用 \$service_ensure 这样的变量，这些变量也是 ntp 从 ntp::params 继承过来并本地化的，且有了 default 值。class service 只包含了 ntp 的 service 启停管理。

以下则是关于 module::params 的示例。

ntp/manifests/params.pp 中的代码如下：

```

class ntp::params {

  $autoupdate      = false
  $config_template = 'ntp/ntp.conf.erb'
}

```

```

$disable_monitor = false
$keys_enable     = false
$keys_controlkey = ''
$keys_requestkey = ''
$keys_trusted    = []
$logfile         = undef
$package_ensure  = 'present'
$preferred_servers = []
$service_enable  = true
$service_ensure  = 'running'
$service_manage  = true
.
.
.
$default_config      = '/etc/ntp.conf'
$default_keys_file   = '/etc/ntp/keys'
$default_driftfile    = '/var/lib/ntp/drift'
$default_package_name = ['ntp']
$default_service_name = 'ntpd'

case $::osfamily {
.
.
.
'RedHat': {
    $config      = $default_config
    $keys_file   = $default_keys_file
    $driftfile    = $default_driftfile
    $package_name = $default_package_name
    $service_name = $default_service_name

    $restrict = [
        'default kod nomodify notrap nopeer noquery',
        '-6 default kod nomodify notrap nopeer noquery',
        '127.0.0.1',
        '-6 ::1',
    ]
    $iburst_enable = false
    $servers       = [
        '0.centos.pool.ntp.org',
        '1.centos.pool.ntp.org',
        '2.centos.pool.ntp.org',
    ]
}
.
.
.
}
}

```

class params 没有任何继承，它被 ntp 所继承的。可以看出，所有可定义参数都在

其中，而且还根据不同 os 进行了配置分类。因此，维护一个社区的模块其实是一个很费时的事情。所以再次感谢开源的力量！一般情况下不建议直接更改 params 里的值，而是用 site.pp、enc 或 hiera 更改默认 parameter。

site.pp 中更改 parameter 的示例如下：

```
node agent1 {
  class { 'ntp':
    autoupdate => true,
  }
}
```

(2) Parameters 设计

一般来说参数设计奉行以下几个原则。

第一个原则，采用统一的命名规范。

- ❑ 以事物_属性 (thing_property) 的方式来命名，比如 package_ensure、service_enable。
- ❑ 如果是一系列参数集合的控制参数，那么大多数情况它是一个布尔值 true or false，可以用事物_manage (thing_manage) 来命名，比如 service_manage。

示例代码如下：

```
if $service_manage == true {
  service { 'ntp':
    ensure    => $service_ensure,
    enable    => $service_enable,
    name      => $service_name,
    hasstatus => true,
    hasrestart => true,
  }
}
```

第二个原则，尽可能不 hard coded。

笔者也写过很多 hard coded，写的时候爽，到后面就发现一堆写死的东西藏在某个 file 或者 template 里面，最后整个代码变得非常不灵活，又容易给其他工程师带来很多坑。最后，陆陆续续花费了团队 1 个月的时间来整理这些 hard coded 的代码，以实现参数 (parameter) 化。

第三个原则，考虑 site.pp/enc/hiera 等中央配置管理。

中央配置管理的重要性不言而喻，比如需要重新在一个新的 IDC 搭建一个新的项目，就会发现大量 parameter 要改，如果没有中央配置管理，就只能在模块文件里一个个地去找，说不定日后还发现不少坑。

当然有读者会说，如果那么多 parameter 都放在 site.pp 中，这个文件不是巨大无比啊？因此这时候需要一个 enc 或者 hiera，后文会详解 enc。

第三步，模块测试。

下面介绍几种测试过程。

(1) 语法测试

针对 pp 文件的 syntax 测试如下:

```
puppet parser validate xx.pp
```

针对 erb 文件的 syntax 测试如下:

```
erb -x -T '-' xx.erb | ruby -c
```

(2) 不同的 environment 测试

environment 会在后文详解, 这里介绍理念。一个项目最好有 test、dev、production 三个 environment。test 是自己的试验机所在的 environment; dev 是代码的开发环境所在的 environment; production, 顾名思义, 即生产系统所在的运行环境。在模块测试过程中, 至少可以在放入生产环境前跑 2 遍, 自己试验机可以过滤掉大部分的 bug 或 unexpect, dev environment 可以捕捉漏网之鱼 (比如开发帮忙验证网站功能), 最后才能放心地投入到生产环境中去。

(3) rspec-puppet 测试

上文说过 rspec-puppet 是 Puppet 的标准 unit test, 想成为 module 大师的可以自行研究。

10.1.2 使用公有模块

上节分析了模块目录结构的最佳实践, 有了一定的基础后就可以读懂社区的公有模块了, 这节就来分析下如果获得更多的模块, 以及选择模块的小技巧。

1. 基本功能

Puppet 公有模块管理就好比 yum 管理 rpm 包, 也有 install、uninstall、upgrade、list、search 等, 模块仓库地址为 <https://forge.puppetlabs.com/>。当然可以直接上 forge 网站上查询和下载, 也可以通过 puppet module 子句直接用命令行来管理。

Puppet module 有如下参数需要了解。

- ❑ build: 打包一个符合 forge 规范的模块目录, 太高端, 笔者还未做深入研究。
- ❑ changes: 显示已安装的模块中被人为修改过的文件, 照理说所有普通文件应该是只读的 (find ntp/ -type f | xargs ls -l), Puppet 官方希望大家都是通过 site.pp/enc/hiera 的方式中央管理, 如果项目中还是有人乱改, 用这个参数可以找出哪里被改了。
- ❑ generate: 生产一个模块的样板, 该样板符合 forge 的规范, 但由于笔者还不是大师, 所以基本没使用过。
- ❑ install: 安装。
- ❑ list: 列出已安装的模块。
- ❑ search: 在 forge 上搜索模块。
- ❑ uninstall: 卸载模块。
- ❑ upgrade: 升级模块。

2. 选择模块

(1) approved VS. supported

forge 上的 Puppet 公有模块有 2 大类：第一类是 supported 模块；第二类是 approved 模块。

supported 模块的特点如下：

- ☐ 官方认证并维护。
- ☐ 24*7 的商业支持。
- ☐ 强大的社区支持，可以通过多种渠道获得帮助，比如论坛发帖、opensource maillist 和 irc.freenode.net 中的专属 channel 等。
- ☐ 官方模块的命名前缀为 puppetlabs。
- ☐ 只有 20 多个。

approved 模块的特点如下：

- ☐ 官方认证。
- ☐ 强大社区支持，也可以通过多种渠道获得帮助，比如 forge 的模块页面的 'Report Issues' link、Puppet Users Google Group、ask.puppetlabs.com 和 irc.freenode.net 中的专属 channel 等。
- ☐ 非官方模块的前缀不是 puppetlabs。
- ☐ 有 3000 多个。

那么，如何选择好的 approved 模块呢？

根据上面的对比，读者肯定可以看出，很多常见模块其实还是要靠开源社区的力量。当然好的开源社区一定是活跃的社区，现在就来介绍，如何找到那些流行模块。

- ☐ 用 <https://forge.puppetlabs.com> 来查找模块。好处是直观，可以看出模块下载数来判断该模块的流行程度，还可以点击维护者查看该维护者所有的模块多不多，以及其他模块的下载数情况等。比如 example42/nginx，可以点击 example42 看到他有很多其他很流行的模块。
- ☐ 直接搜索自己熟悉的优秀的维护者看他有没有该模块。比如前文提到的 example42，应该是最大的模块提供者了。还有其他的比如 evenup/camptocamp/theforeman，较 example42 小，不过也初具规模了。
- ☐ 还有一种是走精品不走量的模块。比如 jfryman/nginx，下载数量远远超过其他著名维护者的 Nginx 模块。至于到底用哪一个就见仁见智了。

(2) 管理模块

其实管理模块前文已陆陆续续提到，这里总结如下：

- ☐ 尽量不更改模块内容。
- ☐ 使用 site.pp/enc/hiera 等中央管理工具更改模块的 parameter 来实现个性化。
- ☐ 优选官方 puppetlabs 模块，或优秀维护者的 approved 模块。如果还是不满足，比

如软件太小众，那么写自己的模块。

除了这些，再介绍一下如何读懂模块，毕竟管理好的前提是完全了解该模块。

- ❑ 直接读源代码。适合高级玩家。
- ❑ 看模块下的 README.markdown 和 tests 目录下的 example。适合中级玩家。
- ❑ 看模块在 forge 上的主页。比如 <https://forge.puppetlabs.com/example42/nginx>。适合中初级玩家或 html 爱好者，笔者是“视觉动物”，如果 doc 不是五颜六色且要滚动 3 屏以上，真是没法看☺。

10.1.3 神奇的 enc

1. 什么是 enc

enc 全称 external node classifier，翻译为“外部的节点分类器”，顾名思义就是一个在 Puppet 之外的 node 定义 class，它是一个脚本可以告诉 Puppet master 这个 node 应该有哪些 class，因此，它可以替换节点在 site.pp 的定义，这样即可扩展 Puppet，使其连接其他外部系统（例如一个 CMDB），还可以避免项目增长后，site.pp 变成一个又臭又长、难以维护的超长文件。enc 具有如下特点：

- ❑ enc 是在 Puppet master 上可执行的脚本，而且不一定要用 Ruby 来写。
- ❑ 该脚本只接受一个参数，node 的 fqdn。
- ❑ 该脚本的 output 必须用一个 YAML 输出来描述这个 node 应该有的内容，并且要符合相关规范。
- ❑ 可以和 site.pp 连用，node 在 enc 和 site.pp 中的定义会进行有机合并（重复定义的部分会被 enc 覆盖）。

2. enc 和 site.pp 的区别

enc 和 site.pp 的区别见表 10-1。

表 10-1 enc 和 site.pp 的区别

	enc	site.pp
功能	declare 各个 node 的 class；定义一个 node 的 global parameter，可以在该 node 的每个 class 中使用；它还能定义一个 node 所属的 environment	除了 enc 的三个基本功能外，可以像一个正常的 Puppet 的 pp 文件定义 resource、写条件语句等。但如果在撰写 Puppet 时遵循原子性，并且对于模块中的 parameter 做得相当到位，那么 site.pp 的其他功能就显得很鸡肋了
行为	在 puppet master compile 一个 node 的 Puppet 代码的时候，它只会传 node 一个参数，该 node 的 fqdn，然后得到 enc 的输出并把相关 class 定义，parameter 定义和 environment 定义代入相应 Puppet 模块，一起 compile 成 catalog 传给该 node 的 Puppet agent 执行。另外需要注意的是，任何 enc 中的设置都将覆盖任何 puppet conf 中的设置，包括 site.pp	在 puppet master compile 一个 node 的 Puppet 代码的时候，它会先尝试 fqdn 然后逐级尝试短域名 (agent1.example.com->agent1.example->agent1)，然后到 site.pp 的相应的 node 定义区域，执行 Puppet 代码，当然不仅包括 class/parameter/environment 定义，还包括任何符合 Puppet 语言规范的代码

(续)

	enc	site.pp
受众	有一个中央 CMDB 或者类似系统的项目，比如可以使用 zabbix 的 inventory 系统，并通过 zabbix api 制作自己的 enc	没有中央 CMDB 的项目，且未来也不会有，由于服务器数量不会超过百台，维护 CMDB 成本比 site.pp 大得多

3. 配置 enc

配置命令如下：

```
[master]
node_terminus = exec
external_nodes = /usr/local/bin/my_enc.py
```

node_terminus 是 external_nodes 的前置参数，默认是 plain，改成 exec 是告诉 Puppet master 请执行 external_nodes 定义的脚本路径，以获得 compile 过程中 node 所需要的定义。external_nodes 是脚本路径，上文提过，可以是任意语言写的可执行脚本，对于 Puppet master 必须要有可执行权限。

4. 如何撰写自己的 enc

上文提到过，enc 的 output 必须包括 classes、parameters、environment 三个主 key 的 YAML 输出格式，以下是一个简单的 output：

```
[root@puppet /]# /usr/local/bin/my_enc.py agent1.example.com

---
classes:
  common:
  ntp:
    ntpserver: 0.pool.ntp.org
parameters:
  zabbix_server: zabbix.example.com
  iburst: true
environment: production
```

在上述代码中，classes 为该 node 声明了 common 和 ntp 两个 class，并且给 ntp class 传参 ntpserver，值为 0.pool.ntp.org。

对于 parameters，这里定义了两个 global 的 parameter，即 zabbix_server 和 iburst，前者 common 模块会用到，后者 ntp 会用到。其实在 classes 中定义相应参数是规范做法，在 parameters 中定义则是懒人做法，这样一来，就不用在 class 中定义需要传的参数了，在模块代码里可以直接使用已经在 parameters 中定义过的变量。

代码中还定义了这个节点所属的环境，environment 的作用是分隔了不同环境的 manifest（即 site.pp）和 modulepath，用于测试、隔离等，后会详解使用方式。

至于 my_enc.py 应该长什么样，这里给出一个用 zabbix inventory 所做的例子。

```

[root@puppet /]# pip install PyYAML -i http://pypi.douban.com/simple/
[root@puppet /]# pip install pyzabbix -i http://pypi.douban.com/simple/
[root@puppet /]# cat /usr/local/bin/my_enc.py
from pyzabbix import ZabbixAPI
import sys
import re
import yaml

zapi = ZabbixAPI("http://zabbix.example.com/zabbix")
zapi.login("admin", "zabbix")
# print "Connected to Zabbix API Version %s" % zapi.api_version()

outcome={"classes": {}, "parameters": {}, "environment": "test"}
host = sys.argv[1]

zbx_get_result = zapi.host.get(output="extend", withInventory="true", select-
    Inventory="extend", filter={"host": host})
if zbx_get_result:
    h = zbx_get_result[0]
    for i in h["inventory"]:
        if h["inventory"][i]:
            if i == "tag":
                outcome["environment"] = h["inventory"][i]
            elif re.match("software_app_[abcde]", i):
                outcome["classes"][h["inventory"][i]] = []
            else:
                outcome["parameters"][i] = h["inventory"][i]

print yaml.safe_dump(outcome, default_flow_style=False)

```

这里用到了开源的 pyzabbix。Zabbix 的 api 和 inventory 的使用方法不是本章的重点，请读者花时间研究。此外，这里使用了 tag 这个 inventory 属性作为 Puppet 中 environment 的映射，software_app_a~software_app_e 作为 Puppet 中 class 的映射，最后以漂亮的 yaml 格式输出。

当然 Zabbix 的 inventory 特性的好处是可以映射为一个监控的 item 实时更新，而且有 UI 和 search 支持，不过使用的前提是对 Zabbix 有一定的了解。使用其他系统作为 enc 的来源其实也一样，需要规划每个属性所要映射到 Puppet 的定义，并且了解其 API 懂得如何去调用，这些功夫虽然花时间，然而一旦完成，后期的收益是无法估量的，也是迈向 DevOps 之路的重要一步。

10.1.4 自定义 resource type/facter/function

自定义系列是 Puppet 提供给广大使用者最自由的功能，可以用在各种奇奇怪怪的场景，发挥出巨大的作用，但也是入门最难的功能，因为它需要使用者有一定的 Ruby 使用技巧和编程功底。本章主要专注于 Puppet，并非深入 Ruby，因此介绍些编写规范和给出一些代码示例，点到为止，主要是给读者一点自定义 resource type/facter/function 的概念和使用

方法，因为这对于 99% 的场景来说已经足够。而且，Puppet 官方也考虑到这一点，特地维护了 Puppet forge 这个非常好的社区模块分享站点，从而让不擅长编程的 ops 找到了福音，直接使用编程达人们分享模块里的已经做好的自定义 resource type/facter/function。

1. 自定义 resource type

这里以一个例子直接来看 resource type 的结构，该结构包含 2 个文件：type 和 provider，代码如下：

```
[root@puppet modules/]# find sysctl/lib/ -type f
sysctl/lib/puppet/provider/sysctl/sysctl.rb
sysctl/lib/puppet/type/sysctl.rb
```

type 文件放在 \$module_name/lib/puppet/type/<TYPE NAME>.rb 里。provider 文件放在 \$module_name/lib/puppet/provider/<TYPE NAME>/<PROVIDER NAME>.rb 里。该示例中 provider name 也是 sysctl，当然可以换个名字（比如 linux_sysctl），但是使用起来不方便，如果只有一个 provider，没必要给自己找麻烦，原因参考下文。

下面是使用示例：

```
[root@puppet modules/]# vim sysctl/manifests/init.pp
sysctl { "fs.file-max": value => "100000" }
```

如果在上文已将 provider name 改成 linux_sysctl，这里就要写 sysctl { "fs.file-max": value => "100000", provider => "linux_sysctl" }。不过，放着好好的 default 值不用，是不是有点画蛇添足呢？

下面来揭晓代码的庐山真面目，读者看看就好，想研究 Ruby 的可以自行解读。

```
[root@puppet modules/]# cat sysctl/lib/puppet/provider/sysctl/sysctl.rb
require 'puppet/provider/parsedfile'
```

```
Puppet::Type.type(:sysctl).provide(:sysctl,
  :parent => Puppet::Provider::ParsedFile,
  :default_target => "/etc/sysctl.conf",
  :filetype => :flat) do

  desc "Puppet provider for Linux sysctls"

  confine :exists => "/etc/sysctl.conf"
  text_line :comment, :match => /^#/
  text_line :blank, :match => /\s*$/

  record_line :sysctl, :fields => %w{name value}, :separator => /\s*=\s*/,
    :block_eval => :instance do
    def post_parse(record)
      record
    end
  end
end
```

```

        def to_line(record)
            return "%s = %s" % [record[:name], record[:value]]
        end
    end
end

[root@puppet modules/]# cat sysctl/lib/puppet/type/sysctl.rb
Puppet::Type.newtype(:sysctl) do
    @doc = "Linux sysctl Puppet type"

    ensurable

    newparam(:name) do
        desc "The name of the sysctl"

        isnamevar
    end

    newproperty(:value) do
        desc "The value for the sysctl"
    end
end
end

```

2. 自定义 factor

自定义 factor 的好处是，可以在每台机器上执行命令得出该机器特殊的属性。这里给出的示例是查出机器是无盘还是有盘，因为在项目中无盘和有盘是共存的，因此需要判断不同的情况下如何写 Puppet。

先来看它的结构，这里只有 1 个文件，即一个 factor 文件，代码如下：

```

[root@puppet modules/]# find site-default/lib/ | grep rootfs
site-default/lib/factor/rootfs_type.rb

```

由于 factor 是 agent 从 server 获得代码执行的结果，因此要在 agent 重新拉取 master 的配置，才可以看到结果，使用命令如下：

```

[root@diskless_agent /]# puppet agent -t
[root@diskless_agent /]# df /
Filesystem            1K-blocks      Used Available Use% Mounted on
-                     2460672    1461424    874248   63% /
[root@diskless_agent /]# factor -p | grep rootfs
rootfs_type => -

```

```

[root@normal_agent /]# puppet agent -t
[root@normal_agent /]# factor -p | grep rootfs
rootfs_type => ext4

```

```

[root@puppet modules/]# vim syslog-ng/manifests/init.pp
if $rootfs_type == '-'{
    $syslog_conf = not_local_copy.conf.erb
}

```

```

} else {
    $syslog_conf = local_copy.conf.erb
}

file { ["/etc/syslog-ng.conf":
    content => template("syslog-ng/$syslog_conf"),
    require => Package["syslog-ng"],
    notify => Service["syslog-ng"]
}

```

可以看到 diskless 的 rootfs_type 是 -, 而普通 agent 的是 ext4。rootfs_type 可以用在任何 Puppet 模块里, 这里用到了 syslog 里, 无盘没有空间, 因此 syslog 配置用一个不存本地的配置 not_local_copy.conf.erb。

现在来揭晓代码的庐山真面目, 这个比较简单, facter 应该需要自定义的场景最多的情况, 也算是福音。

```

[root@puppet modules/]# cat site-default/lib/facter/rootfs_type.rb
Facter.add("rootfs_type") do
    confine :kernel => "Linux"
    setcode do
        Facter::Util::Resolution.exec('df -PT / | awk \'/\|/ {print $2}\'' )
    end
end

```

如果要使用上述代码, 前半段可以照抄, 因为大多数情况可以调用 shell 直接完成, 因此比较简单。注意转义符 \ 的用法, 这里分别转义了 awk 中的 2 个单引号, 当然也转义了需要匹配的正斜杠 '/' 字符。

3. 自定义 function

这里给一个场景, Puppet 代码中时不时会遇到要定义 password 的时候, 但是如果直接明文写, 以后要进行 Puppet 代码版本管理的时候, 这些 production 的密码也会被传上 git 仓库, 是非常不安全的, 因此这些 password 最好存放于本地的另外一个地方, 不在 modules 目录下, 但是可以通过函数在 Puppet 代码中调用, 这样就安全了。

先来看看它的结构, 这里只包含了 1 个文件, 即一个 function 文件。代码如下:

```

[root@puppet modules/]# find site-default/lib/ | grep fetch_passwd
site-default/lib/puppet/parser/functions/fetch_passwd.rb

```

下面是使用方式:

```

[root@puppet modules/]# vim mysql/manifests/init.pp
$monitor_db_password = fetch_passwd("monitor_pw")
$production_app_db_password = fetch_passwd("app_pw", "production")

[root@puppet modules/]# vim /etc/secret.json
{
    "site_wide" : {

```

```

    "monitor_pw": "password_for_monitor",
  },
  "production": {
    "app_pw": "very_complex_password_for_app",
  },
}

```

site_wide 的 tree 下，是 fetch_passwd 不加参数的情况；production 的 tree 下，是 fetch_passwd 加了参数 "production" 情况。

现在来揭晓代码的庐山真面目，这个稍微有点复杂，建议读者边学 Ruby，边理解。

```

[root@puppet modules/]# cat site-default/lib/puppet/parser/functions/fetch_passwd.rb
require 'rubygems'
require 'json/pure'

```

```

module Puppet::Parser::Functions
  newfunction(:fetch_passwd, :type => :rvalue) do |args|
    if args.size < 1 or args.size > 2 then
      raise Puppet::ParseError, "This function takes either one or two
        arguments"
    end

    passwordName = args[0]
    envName = args[1]

    if envName == nil then
      # site wide is defaults
      envName = "site_wide"
    end

    passwordFile = File.open('/etc/secret.json', 'r')
    passwordJSON = passwordFile.read
    passwordFile.close

    passwords = JSON.parse(passwordJSON)

    if passwords.has_key?(envName) then
      if passwords[envName].has_key?(passwordName) then
        return passwords[envName][passwordName]
      else
        raise Puppet::ParseError, "\"#{envName}\" missing key for \"#{passwordName}\""
      end
    else
      raise Puppet::ParseError, "Panic, Even the default site wide ENV
        doesn't exist"
    end
  end
end
end

```

10.2 管理好一个 Puppet 集群

10.2.1 监控 Puppet 运行状况

监控是一个运维保证线上 service 运行状况的根本，通常，思考对于什么服务应当监控什么是一个运维的基本功，也是一个考量运维价值的重要因素，因为及时发现并处理，就意味着减少 down time，对于业务来说就是避免更大的损失，体现了运维的价值。

下面将从两个方面来思考。

1. 可用性

用通俗的话来说，可用性就是在不好用的时候能够及时探测出来，因此，我们要考虑以下问题：

- ❑ 这个服务对于业务的意义是什么？
- ❑ 对于服务的 client 来说，有哪些错误类型和症状？
- ❑ 对于服务本身来说，出错的时候，有哪些症状？

可以进行错误类型的枚举来制定相应的监控项，现在就以上述的问题来对 Puppet 进行分析。

第一个问题，Puppet 对于业务的意义是什么？

答：保证 Puppet master 上的代码可以及时应用到 Puppet agent 上，以达到中央管理的目的。

第二个问题，对于 Puppet agent 来说，有哪些错误类型和症状？

答：基于 Puppet 的对于业务的意义来说，以下错误都是不可接受的。

- ❑ 从 master 上获得的 catalog 无法执行，有 error/warning。
- ❑ 根本无法和 master 进行通信，获取 catalog 都失败了。
- ❑ Puppet agent 本身都没在运行，或者被人为地 disabled 了。

那么这些错误的症状是什么呢？Puppet 为我们准备了如下三个文件来指出上述错误。

/var/lib/puppet/state/last_run_summary.yaml 中的代码如下：

```
---
version:
  config: 1432993826
  puppet: "3.7.5"
changes:
  total: 9
time:
  last_run: 1432993930
  service: 11.178494
  package: 57.138766
  anchor: 0.00043
  config_retrieval: 25.1438100337982
  file: 5.738155
```

```

total: 100.474168033798
filebucket: 0.000259
notify: 0.001315
exec: 1.271416
schedule: 0.001523
resources:
  scheduled: 0
  restarted: 0
  changed: 9
  total: 31
  failed: 1
  failed_to_restart: 0
  skipped: 0
  out_of_sync: 10
events:
  success: 9
  failure: 1
  total: 10

```

根据 events 里面的 failure 个数，可以看出从 master 上获得的 catalog 是否执行时有 error/warning。根据 time 里的 last_run，可以看出是否有长时间无法从 master 上获得 catalog 的情况。

/var/lib/puppet/state/last_run_report.yaml 中的代码如下：

```

- !ruby/object:Puppet::Util::Log
  time: 2015-05-30 13:50:56.656213 +00:00
  source: /Stage[main]/Motd/File[/tmp/ds/2]
  level: !ruby/sym debug
  message: "The container /tmp/ds will propagate my refresh event"
  tags:
    - class
    - default
    - file
    - hell1
    - hell2
    - node
    - debug
    - motd
- !ruby/object:Puppet::Util::Log
  time: 2015-05-30 13:51:53.828871 +00:00
  source: Puppet
  level: !ruby/sym err
  message: "Execution of '/usr/bin/yum -d 0 -e 0 -y list dstat.x86_64' returned
    1: Error: No matching Packages to list"
  tags:
    - err

```

在上述代码中，从 level 这个 key 可以分析出具体 error 的 message。

/var/lib/puppet/state/agent_disabled.lock 中的代码如下：


```
[root@agent1 /]# puppet agent --disable "for testing"
```

上述命令是 disable puppet agent 的命令，会产生相应的 /var/lib/puppet/state/agent_disabled.lock。

```
[root@agent1 /]# cat /var/lib/puppet/state/agent_disabled.lock
{"disabled_message":"for testing"}
```

知道症状以后，接下来就是实现监控了，这里不再展开，想必读者都有各自的监控软件，可以自己写相应的监控脚本。如果是用 Nagios，可以去 <https://exchange.nagios.org> 上查找别人写的一些脚本作为参考；如果是 Zabbix，可以使用 GitHub 上一些分享，如 <https://github.com/shamil/puppet-zabbix-reports>。

这里再提一个比较好的 aggregation 工具来统一查看 Puppet agent 的运行状况——Puppet Dashboard。关于如何搭建 Puppet Dashboard，后文会提及，这里主要介绍思路，比如对于 Puppet，你的项目目前只是处于试水状态，并不想各种服务器 Puppet 报警充斥介绍监控面板的时候，可以使用 Puppet Dashboard 来得到一个聚合报告，并进行整体报警。当然笔者建议是以两者配合不同警报级别结合的方式去更好地展现 Puppet agent 的运行状况。这里给出一个使用 Zabbix 作为监控系统的例子：<https://github.com/simonswine/zabbix-puppetdashboard>。

第三个问题，对于 Puppet master 来说，出错的时候有哪些症状？

上文提到了 Puppet agent 出错的状况，当然如果所有的 Puppet agent 都无法从 Puppet master 上拿到 catalog，那也代表了 Puppet master 工作不正常。当然我们有更直接地从 master 上进行监控的方式，即使用 Puppet master 的 status api。

下面代码用于添加 api 权限。

```
[root@puppet /]# /etc/puppet/auth.conf

path /status
auth any
allow *

# 注意要在下面这条deny all之前定义权限
# deny everything else; this ACL is not strictly necessary, but
# illustrates the default policy.
path /
auth any
```

以下是访问 status api 的命令：

```
[root@puppet /]# curl --cacert /var/lib/puppet/ssl/certs/ca.pem --cert /var/lib/puppet/ssl/certs/`facter fqdn`.pem --key /var/lib/puppet/ssl/private_keys/`facter fqdn`.pem -H 'Accept: pson' https://puppet:8140//production/status/test
{"version":"3.7.5","is_alive":true}
```

2. 性能

同样，对于性能监控这里也有三个相应的问题。

- ❑ 对于服务的 client 来说，慢的症状是什么？
- ❑ 对于服务本身来说，慢的症状是什么？
- ❑ 对于服务本身来说，是否有将要变慢的症状？（预警）

针对第一个问题，对于 Puppet agent 来说，慢的症状是运行一次时间过长。来看个示例。

`/var/lib/puppet/state/last_run_summary.yaml` 中的代码如下：

```
---
changes:
  total: 4
version:
  puppet: "3.7.5"
  config: 1433059415
events:
  total: 8
  success: 4
  failure: 4
time:
  total: 29.148274868927
  filebucket: 0.000151
  file: 0.703988
  notify: 0.001447
  last_run: 1433059970
  exec: 0.127579
  service: 0.080962
  schedule: 0.001158
  package: 25.01855
  config_retrieval: 3.214439868927
```

`time` 下的 `total` 是总运行时长，当然也有一些各个部分基本耗时统计，比如 `file` 非常长的话，可以考虑 Puppet 的 `file service` 性能不够，对此，建议单独分一台的 `file server` 为所有 `agent` 服务。

`/var/lib/puppet/state/last_run_report.yaml` 中的代码如下：

```
[root@puppet /]# cat /usr/local/bin/analyze_puppet_slow.py
import yaml

def construct_ruby_object(loader, suffix, node):
    return loader.construct_yaml_map(node)

def construct_ruby_sym(loader, node):
    return loader.construct_yaml_str(node)

yaml.add_multi_constructor(u"!ruby/object:", construct_ruby_object)
yaml.add_constructor(u"!ruby/sym", construct_ruby_sym)
```

```

stream = file('/var/lib/puppet/state/last_run_report.yaml','r')
mydata = yaml.load(stream)['resource_statuses']
for i in mydata:
    if 'evaluation_time' in mydata[i]:
        print '%s - %s' % (mydata[i]['evaluation_time'], mydata[i]['resource'])

[root@puppet /]# python /usr/local/bin/analyze_puppet_slow.py | sort -n
.
.
.
0.500516 - Package[dstat2]
1.951594 - Package[mysql]
2.383181 - Package[nagios-plugins-all]
20.177932 - Package[puppetdashboard]

```

同样，可以通过这个详细的报告文件，来得出具体是哪个 resource 慢。

这里的 /usr/local/bin/analyze_puppet_slow.py 是笔者写的一个比较粗糙的分析脚本，不过可用。分析得出上次执行 Puppet agent 慢的原因是安装了 Package[puppetdashboard] 的包，用了 20 多秒，可以得出结论是 Puppet 官方 repo 的速度较慢，可以考虑搭内部的镜像 repo。

针对第二个问题，对于 Puppet master 本身来说，慢的症状是单位 agent 执行时长太过分，即需要监控“总时长 / 总 agent 个数”的值是否在合理范围内。执行时长的获得的方法有 2 种。

(1) Puppet Dashboard

Puppet Dashboard 是一个比较漂亮的 Web 版的 Dashboard，可以让大家直观地查找每个节点的运行情况，包括每步的时间和结果，说白了就是收集了 client 所有的 report，并对其进行了分析。想法很不错，可是不是官方项目，官方支持力度有限，Puppet Dashboard 1.23 放出来 2 年后，再也没出新版本了，而且 master branch 上最后一次 GitHub 提交也是近一年前，所以笔者不是很推荐，不过如果读者有兴趣或要给老板展示漂亮 report，可以自行安装。由于官方在 2 年内已发展到 Puppet 4，但是 Puppet Dashboard 却迟迟不更新，已经将原有相关文档的链接，又重定向回该项目的 GitHub 链接，下面给出两个传送门，方便读者安装。

❑ <https://downloads.puppetlabs.com/docs/dashboardmanual.pdf>

❑ <http://dev.tomatoengine.com/puppet/dashboard/manual/1.2/bootstrapping.html>

(2) Apache 日志

Apache 日志的方法其实很简单，在 Apache 中设置日志格式的代码如下：

```

[root@puppet /]# grep Log /etc/httpd/conf.d/puppetmaster.conf
LogFormat "%h %l %u %t \"%r\" %>s %b %D \"%{Referer}i\" \"%{User-Agent}i\"" puppet
CustomLog /var/log/httpd/puppet.log puppet

```

只要设置是 %D，那么服务器处理本请求所用时间则以微为单位，也就是 1/1000 000 秒。

下面是用脚本分析的方式^①：

① 这个 Perl 脚本出自 <https://gist.github.com/jasonhancock/3439737>，虽然可以用 bash 和 Python 重写，但是还是原文引用了，毕竟是该“大神”原创的思路。

```

[root@puppet /]# cat analyze.pl
#!/usr/bin/perl

use strict;
use warnings;
use Data::Dumper;

my $file = $ARGV[0] or die('Must pass an apache access logfile');

open IN, "<$file" or die("Can't open $file");

my %indirectors;

while(my $line=<IN>) {
    chomp($line);
    my @pieces = split(/\s+/, $line);
    # print Dumper(\@pieces);
    my $method = $pieces[5];
    $method=~s/"//g;
    my ($blah,$env, $indirector, $resource) = split(/\//, $pieces[6], 4);
    #print "$env|$indirector|$resource\n";
    my $ms = $pieces[10]/1000; # convert to milliseconds

    my $key = sprintf('%-4s %s', $method, $indirector);
    $indirectors{$key}{'count'}++;
    $indirectors{$key}{'sum'}+= $ms;
    push(@{$indirectors{$key}{'raw'}}, $ms);
    $indirectors{$key}{'min'} = $ms if(!defined($indirectors{$key}{'min'})) || $ms
        < $indirectors{$key}{'min'};
    $indirectors{$key}{'max'} = $ms if(!defined($indirectors{$key}{'max'})) || $ms
        > $indirectors{$key}{'max'};
}

print "Counts Per Indirector:\n";
foreach my $indirector(sort {@indirectors{$b}{'count'} <=> $indirectors{$a}
    {'count'}} keys %indirectors) {
    printf(" %7d %s\n", $indirectors{$indirector}{'count'}, $indirector);
}

print "\n\n";
print "Statistics by Indirector (milliseconds):\n";

printf(" %-25s %7s %7s %7s %7s\n", '', 'AVERAGE', 'STD_DEV', 'MIN', 'MAX');
foreach my $indirector(sort keys %indirectors) {
    $indirectors{$indirector}{'avg'} = $indirectors{$indirector}{'sum'}/$indirec
        tors{$indirector}{'count'};

    my $sum_squares=0;
    foreach my $value(@{$indirectors{$indirector}{'raw'}}) {
        $sum_squares += ($value - $indirectors{$indirector}{'avg'})**2;
    }
}

```

```

    }

    $indirectors{$indirector}{'stddev'} = ($indirectors{$indirector}{'count'} -
        1) == 0
        ? 0
        : sqrt($sum_squares / ($indirectors{$indirector}{'count'} - 1));

    printf("  %-25s %7d %7d %7d %7d\n",
        $indirector,
        $indirectors{$indirector}{'avg'},
        $indirectors{$indirector}{'stddev'},
        $indirectors{$indirector}{'min'},
        $indirectors{$indirector}{'max'})
};
}

```

```
[root@puppet /]# perl analyze.pl /var/log/httpd/puppet.log
```

Counts Per Indirector:

```

4166 GET   file_content
2934 GET   file_metadatas
1467 POST  catalog
1133 GET   node
1133 PUT   report
  24 GET   file_metadata
  21
  9 GET   certificate
  1 GET   certificate_request
  1 PUT   certificate_request
  1 GET   certificate_revocation_list

```

Statistics by Indirector (milliseconds):

	AVERAGE	STD_DEV	MIN	MAX
	0	0	0	0
GET certificate	1977	2610	2	6049
GET certificate_request	27	0	27	27
GET certificate_revocation_list	4	0	4	4
GET file_content	7	7	2	70
GET file_metadata	647	455	2	1044
GET file_metadatas	57	81	2	1057
GET node	872	1661	3	51658
POST catalog	1446	789	48	9232
PUT certificate_request	5048	0	5048	5048
PUT report	1018	1299	42	7556

可以看出，该脚本帮忙分析了无误 cert、passenger、report，还是 Apache 是拉取 file 的瓶颈，这台 Puppet server 的瓶颈主要在于 cert 和 passenger compile catalog，可以考虑采用 CA 和水平扩容的方式来缓解，在后文 Puppet server 端优化会详解。

针对第三个问题，对于 Puppet master 本身来说，是否有将要变慢的症状。这要从三种模式的不同症状分别来分析。

一种是 WEBRick 模式，对此就不要纠结了，这种模式本身就非常低效的，放弃吧！

第二种是 Passenger 模式，下节会提到如何使用它，用户所要监控的就是 Passenger 的 worker 状况。

示例代码如下：

```
[root@puppet /]# passenger-status
----- General information -----
max          = 36
count        = 9
active       = 2
inactive     = 7
Waiting on global queue: 0

----- Application groups -----
/etc/puppet/rack:
  App root: /etc/puppet/rack
  * PID: 11105   Sessions: 0   Processed: 261   Uptime: 2m 23s
  * PID: 10856   Sessions: 0   Processed: 488   Uptime: 2m 49s
  * PID: 7518    Sessions: 0   Processed: 758   Uptime: 8m 28s
  * PID: 10556   Sessions: 0   Processed: 404   Uptime: 3m 45s
  * PID: 10461   Sessions: 0   Processed: 383   Uptime: 4m 13s
  * PID: 11102   Sessions: 0   Processed: 50    Uptime: 2m 23s
  * PID: 10978   Sessions: 0   Processed: 294   Uptime: 2m 35s
  * PID: 11108   Sessions: 1   Processed: 60    Uptime: 2m 23s
  * PID: 7687    Sessions: 1   Processed: 786   Uptime: 8m 17s
```

可以看出，36 个 worker 完全足够用，如果 active 长期处于 30+ 的状态，且 CPU 使用率很高，那么可以考虑水平扩容了。其实 worker 也不是越多越好，还要结合上文 apache log 的分析脚本，毕竟单位时间内处理的请求才是衡量性能的标准。如果开了 100 个 worker，1 分钟内完成了 3000 个请求，平均每个请求平均耗时 5 秒钟，那么对比 30 个 worker，1 分钟内完成了 4500 个请求，每个请求平均耗时 0.4 秒钟，用户肯定会使用后者的 worker 配置，从而避免过多 worker 的 contex switch 和无效占用带来的系统开销。

第三种模式是 Puppetserver。下节同样要提到如何使用它，用户所要监控的就是 Puppetserver 的 worker 和 heapsize 状况。

对 Puppetserver 的监控，熟悉 Java 的同学肯定会想 jvm 的监控，包括 jps、jinfo、jstat、jmap、jconsole 等一大堆。当然，这里需要用的只是监控 heapsize 状况，以下命令即可实现。

```
[root@puppet /]# sudo -u puppet jps
25062 Jps
16838 main
[root@puppet /]# sudo -u puppet jinfo -flag MaxPermSize 16838
-XX:MaxPermSize=268435456
[root@puppet /]# sudo -u puppet jinfo -flag PermSize 16838
-XX:PermSize=21757952
```

这里可以看到，由于做实验的 Puppet 代码不多，Permsize 只有 21MB，但是 Max-PermSize 却配了 256MB，内存使用率就会是 256MB*max-active-instances，非常浪费，可以适当减小 MaxPermSize。

和 Passenger 模式一样，用户必须分析 log，Puppetserver 的 log 定义在 /etc/puppet-server/request-logging.xml，格式和 Apache 雷同，因此，可以使用相同脚本进行分析。

同样，Puppetserver 的 worker 数名叫 max-active-instances，下文会详解，调优思路和 Passenger 类似。

10.2.2 做好 Puppet 的容量规划

1. Puppet 工作流程

在讲解容量规划之前先来介绍 Puppet 的工作流程，如图 10-1 所示。

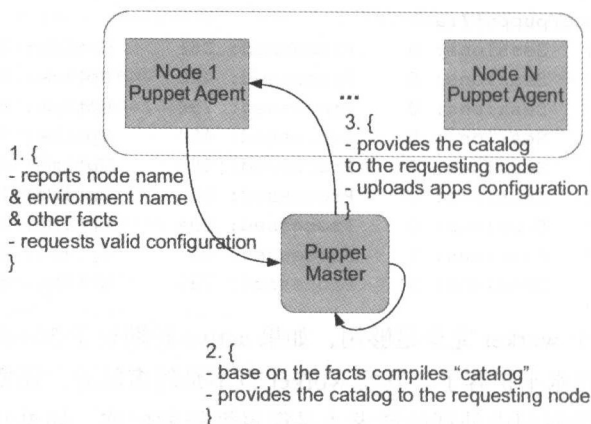


图 10-1 Puppet 的工作流程

第一步，Puppet agent 去往 Puppet master 进行 SSL CA 认证，通告 node name、environment name 和 facts。

第二步，Puppet master 根据 Puppet agent 提供的信息，根据 Puppet 代码，用 ruby compile 成 catalog 再传到 client 去执行。

第三步，Puppet agent 根据 Catalog 获取相关 File 资源，并且在本地执行 Catalog。

整个过程大体是这样的，因此，可以从下面将要讲的几个方面着手，进行性能优化和容量规划。

2. Puppet master 端优化

Puppet master 端的优化，其实有非常多的细节，尤其在代码优化这个领域，但是这不仅需要对每个项目所写的 Puppet 模块进行分析，还需要在 Ruby 上有比较深的造诣，不适合在此展开。因此，笔者将通过介绍“改变 master 运行方式”和“角色分离 & 负载均衡”这两个优化点，来引出 Puppet master 端的优化方向，而对于代码级别的优化，则需要靠读

者平时不断的积累，才能体会。

(1) 改变 master 运行方式

有三种 master 运行方式：一种是 webrick 方式，一种是 passenger 方式，还有一种是 Puppet server 方式，下面分别讲解。

第一种，webrick 方式，webrick 是 Puppet 默认的 ruby 内置的 http 服务，方便快捷，缺点是性能问题，当然如果只有几十台，且没有一起执行的需求（默认 30 分钟跑一次），那么可以继续使用这种方式。

第二种，passenger 方式，passenger 是一款有社区版和商业版之分的 ruby/python/node.js 解析器，社区版虽然功能有限，但应对 Puppet 这样的 admin tool 已经足够。下面给出一个相关示例。

安装 mod_passenger 的命令如下：

```
[root@puppet /]# yum install epel-release
[root@puppet /]# yum install mod_passenger mod_ssl
```

通过以下命令准备 Puppet master 的 rack。

```
[root@puppet /]# mkdir /etc/puppet/rack/{tmp,public} -p
[root@puppet /]# cp `rpm -ql puppet | grep config.ru` /etc/puppet/rack/
[root@puppet /]# chown puppet.puppet -R /etc/puppet/rack/
```

然后添加配置文件 /etc/httpd/conf.d/puppetmaster.conf。

```
Listen 8140
<VirtualHost *:8140>
    SSLEngine on
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:RC4+RSA:+HIGH:+MEDIUM:-LOW
    SSLProtocol all -SSLv2
    SSLCertificateFile      /var/lib/puppet/ssl/certs/puppet.example.com.pem
    SSLCertificateKeyFile    /var/lib/puppet/ssl/private_keys/puppet.example.com.pem
    SSLCertificateChainFile  /var/lib/puppet/ssl/certs/ca.pem
    SSLCACertificateFile    /var/lib/puppet/ssl/certs/ca.pem
    # CRL checking should be enabled; if you have problems with Apache complain-
    # ing about the CRL, disable the next line
    # SSLCARevocationFile    /var/lib/puppet/ssl/ca/ca_crl.pem
    SSLVerifyClient optional
    SSLVerifyDepth 10
    SSLOptions +StdEnvVars

    # The following client headers allow the same configuration to work with Pound.
    RequestHeader set X-SSL-Subject %{SSL_CLIENT_S_DN}e
    RequestHeader set X-Client-DN %{SSL_CLIENT_S_DN}e
    RequestHeader set X-Client-Verify %{SSL_CLIENT_VERIFY}e

    LogFormat "%h %l %u %t \"%r\" %s %b %D \"%{Referer}i\" \"%{User-Agent}i\"" puppet
    CustomLog /var/log/httpd/puppet.log puppet
    # Set this to about 1.5 times the number of CPU cores in your master:
```



```

PassengerMaxPoolSize 36

# Recycle master processes after they service 10000 requests
PassengerMaxRequests 10000

# On some systems where disk I/O is expensive, setting this option to a value
# of x means that the above list of filesystem
# checks will be performed at most once every x seconds. Setting it to a
# value of 0 means that no throttling will take place,
# or in other words, that the above list of filesystem checks will be
# performed on every request.
PassengerStatThrottleRate 120

# Since communication with the puppetmaster from puppetd is a long process
# (more than 20 seconds in most cases)
# and will allow for processes to get recycled better
PassengerUseGlobalQueue on

# The additional Passenger features for apache compatibility are not needed
# with Puppet.
PassengerHighPerformance on

PassengerPoolIdleTime 150

RackAutoDetect Off
RailsAutoDetect Off

RackBaseURI /

<IfModule mod_mem_cache.c>
CacheEnable mem /
CacheDefaultExpire 300
MCacheSize 1024000
MCacheMaxObjectCount 10000
MCacheMinObjectSize 1
MCacheMaxObjectSize 2048000
MCacheRemovalAlgorithm GDSF
CacheIgnoreNoLastMod On
</IfModule>

DocumentRoot /etc/puppet/rack/public
<Directory /etc/puppet/rack>
    Options None
    AllowOverride None
    Order allow,deny
    allow from all
    Options -MultiViews
</Directory>
</VirtualHost>

```

这里给出的是笔者项目中一台 16 核物理机的调优状况，有兴趣的同学可参考 <https://>

www.phusionpassenger.com/documentation/Users%20guide%20Apache.html。passenger 调优这里不再展开。mod_mem_cache 是 Apache 的一个简易内存 cache 系统，可以提高 file resource 相关 request 的效率。

接下来启动 Apache 服务，相当于一个正常的 Puppetmaster 在用了，代码如下：

```
[root@puppet /]# /etc/init.d/httpd start
```

第三种，Puppet server 运行方式，Puppet server 是 Puppet 官方才引进的运行方式，用 Java 进行了重构，可用 JRuby 编译器来运行原有 master 的代码，它完全兼容原有的 Puppet code，由于摒弃了基础版的 passenger，性能有非常大的提升。

安装 Puppetserver 的命令如下：

```
[root@puppet /]# rpm -ivh https://yum.puppetlabs.com/puppetlabs-release-el-6.noarch.rpm
[root@puppet /]# yum install puppetserver
```

启动 Puppet server 的命令如下：

```
[root@puppet /]# /etc/init.d/puppetserver start
```

启动时间会比较长，半分钟左右，接下来就可以像用一个 Puppet master 一样使用它了。

接下去的问题就是如何调优和扩容 Puppet server 了。

调优主要涉及 2 个参数，即 /etc/Puppetserver/conf.d/Puppetserver.conf 里的 max-active-instances 和 /etc/sysconfig/Puppet server 里的 Xms 与 Xmx，它们用来调节 Java 的使用内存，方法和传统开源软件 Apache 一样，也是根据每个 instance 内存使用量来调节 Java 内存分配，并根据测试下来的 CPU 占有率，得出最优的值，具体方法参见 https://docs.puppetlabs.com/puppetserver/1.0/tuning_guide.html。

扩容其实和 passenger 模式类似，公用一个 SSL key，多机器支撑。

(2) 角色分离 & 负载均衡

Puppet master 的 CA 以及 File 服务器分离到不同机器上，就好比 LAMP 应用中 Web 和 db 分离的道理一样。

以下是分离 CA 的步骤。

1) 完成环境搭建，准备如下三台机器。

```
serverA - puppetca
serverB - puppetmaster
serverC - puppetagent
```

这里要搭建好 DNS，或者加入 /etc/hosts，命令如下：

```
serverB_IP puppet
serverA_IP puppetca
```

2) 安装 puppetca，在该过程中，又分为如下一些步骤。

第一步，按照一般的 Puppet master 安装方式搭建 puppetca。

第二步，配置产生 SSL 证书的方式，代码如下：

```
[root@puppetca /]# vim /etc/puppet/puppet.conf
[main]
.
.
.
dns_alt_names = puppetca,puppetca.example.com
certname      = puppetca

[agent]
.
.
.
ca_server     = puppetca

[master]
.
.
.
ca            = true
```

第三步，产生证书。

```
[root@puppetca /]# rm -rf /var/lib/puppet/ssl/
[root@puppetca /]# puppet cert generate --dns_alt_names puppetca.example.com
puppetca
[root@puppetca /]# openssl x509 -in /var/lib/puppet/ssl/certs/puppetca.pem
-inform pem -noout -text | grep DNS
DNS:puppetca, DNS:puppetca.example.com
```

删除老证书的原因是，老证书不是以 puppetca 作为 DNS 产生的。这里其实可以直接通过 puppet cert generate 命令产生基于新 DNS 的证书，因为 puppet.conf 里已经定义妥当，参数写全(--dns_alt_names puppetca.example.com puppetca)只是为了更清晰。此外，最后一条命令只是为了验证一下 dns_alt_names 参数产生的证书，包含所定义的 DNS。

第四步，更改 Apache 配置。

```
[root@puppetca /]# vim /etc/httpd/conf.d/puppetmaster.conf
.
.
.
SSLCertificateFile      /var/lib/puppet/ssl/certs/puppetca.pem
SSLCertificateKeyFile    /var/lib/puppet/ssl/private_keys/puppetca.pem
SSLCertificateChainFile  /var/lib/puppet/ssl/certs/ca.pem
SSLCACertificateFile     /var/lib/puppet/ssl/certs/ca.pem
.
.
.
```

第五步，重启 Apache，使新证书生效。

```
[root@puppetca /]# /etc/init.d/httpd restart
```

3) 安装不认证 SSL 的 Puppet master，这里面包含如下六个步骤。

第一步，按照一般的 Puppet master 安装方式搭建 Puppet master。

第二步，配置产生 SSL 证书的方式，配置文件如下：

```
[root@puppet /]# vim /etc/puppet/puppet.conf
[main]
```

```
.
.
.
dns_alt_names = puppet,puppet.example.com
certname      = puppet
```

```
[agent]
```

```
.
.
.
ca_server      = puppetca
```

```
[master]
```

```
.
.
.
ca              = false
```

这里注意，ca 是 false。

第三步，向 puppetca 申请新证书。

```
[root@puppet /]# rm -rf /var/lib/puppet/ssl/
[root@puppet /]# puppet agent --test --ca_server=puppetca
```

这里删除旧的 SSL 是为了接受 puppetca 的新证书。

第四步，在 puppetca 上颁发证书，命令如下：

```
[root@puppetca /]# puppet cert --list --all
"puppet" (SHA256) E2:B5:CA:51:37:83:2C:85:E1:87:4D:
D9:D0:04:01:AC:36:AC:3A:03:8A:91:2B:51:3E:76:2B:40:CF:F3:0B:B3 (alt names:
"DNS:puppet", "DNS:puppet.example.com")
[root@puppetca /]# puppet cert --allow-dns-alt-names sign puppet
```

从 output 可以看出，正确的 DNS 已经被正确地加入申请。这里签名用了 --allow-dns-alt-names，是为了允许 Puppet master 用自己的名字 Puppet。

第五步，在 Puppet master 上签收证书。

```
[root@puppet /]# puppet agent --test --ca_serve=puppetca
```

签收的命令和申请的相同，签收后会报错，因为 Puppet master 就是本身，还没被正确

启动。

第六步，修改 master 的 apache 证书，并启动 master。

```
[root@puppet /]# vim /etc/httpd/conf.d/puppetmaster.conf
.
.
.
SSLCertificateFile      /var/lib/puppet/ssl/certs/puppet.pem
SSLCertificateKeyFile   /var/lib/puppet/ssl/private_keys/puppet.pem
SSLCertificateChainFile /var/lib/puppet/ssl/certs/ca.pem
SSLCACertificateFile    /var/lib/puppet/ssl/certs/ca.pem
.
.
.

[root@puppet /]# /etc/init.d/httpd start
```

4) Puppet agent 进行 CA 和 master 的分离，这其中包括 4 个步骤，如下。

第一步，配置正确的 puppetca 地址。

```
[root@agent1 /]# vim /etc/puppet/puppet.conf
[agent]
.
.
.
ca_server = puppetca
```

第二步，申请证书。

```
[root@agent1 /]# puppet agent --test --server=puppet --ca_server=puppet
```

命令行参数可以不用写全，因为 puppet.conf 已经配置妥当，这里只是为了更清晰的表达。

第三步，puppetca 颁发证书。

```
[root@puppetca /]# puppet cert sign agent1.example.com
```

如果已经有相当数量的 agent，本次操作是为了分离的架构，那么可以使用 puppet cert sign --all 来简化步骤。

第四步，再次运行，成功！

```
[root@agent1 /]# puppet agent -t
```

讲完了分离 CA 的方法，现在来看看要如何分离 File server。

对于 File server 笔者觉得如果有了 Apache 的 <IfModule mod_mem_cache.c>，分离不是特别必要，而且可以以多 master 的形式水平扩容，因此，这里只是简单提下搭建思路，并不会详细列举步骤。

1) 搭建另外一个 Puppet master 作为 fileserver 用，hostname 为 puppetfileserver。以下代码用于更改该 server 的 /etc/puppet/fileserver.conf。

```
[extra_files]
  path /etc/puppet/modules
  allow *
```

这里的 `/etc/puppet/modules` 要做好同步工作!

2) 改写代码。例如之前是:

```
puppet:///modules/apache/ssl.conf
```

现在是:

```
puppet://puppetfileservers/apache/files/ssl.conf
```

10.2.3 使用版本控制来管理代码

1. 搭建 git

使用 git 应该是系统管理员的基本功, 因为即使不使用 Puppet, 也应该为 ops 脚本和个性化工具进行版本控制, 因此以下只是对 git 点到为止。

安装 git 的命令如下:

```
[root@agent1 /]# yum install git
```

这里使用了 agent1 作为 git server, 而不是 Puppet master, 因为 git 对于 Puppet 代码来说也是一个很好的备份, 装在另外一台机器上较为保险。

初始化 git 的命令如下:

```
[root@agent1 /]# useradd git
[root@agent1 /]# passwd git
[root@agent1 /]# su - git
[git@agent1 /]$ mkdir ~/puppetmodule
[git@agent1 /]$ cd ~/puppetmodule/
[git@agent1 /]$ git --bare init
```

```
[root@puppet /]# cd /etc/puppet/modules
[root@puppet modules]# git clone ssh://git@agent1/home/git/puppetmodule/ modules
```

至此, 一个简单的通过 ssh 控制的 git 已经完成, 让 ldap 接入 ssh, 即可完成通过 ldap 认证的 git 源。注意使用 `--bare`, 这是作为 git center repository 的一个标准, 建立的目录结构和一般 local 的 repository 都不一样。

下面是 git 的基本操作:

```
[root@puppet modules]# vim motd/manifests/init.pp
[root@puppet modules]# git add motd/*
[root@puppet modules]# git status
[root@puppet modules]# git diff motd/manifests/init.pp
[root@puppet modules]# git commit
[root@puppet modules]# git push origin master
```

用 add 再用 commit, 是一个好习惯, 不仅可以 add 多个文件, 而且可以避免 commit

不需要的文件。在 commit 之前，status 要看 commit 的状态，也是一个好习惯，它可以清晰地告诉你如果要 commit，将添加或修改哪些文件。此外，diff 也是一个好习惯，它可以告诉你这次 commit 具体修改了该文件的哪些内容。更多的 git 例子请参考官方文档。

2. 搭建不同 environment

关于 environment，前面 enc 章节已经介绍过，它定义了这个节点所属的环境，environment 的作用是分隔了不同环境的 manifest（即 site.pp）和 modulepath，有测试、隔离等用途。

下面介绍 environments 的设置方法。

```
# 添加一行配置指定environment的path
[root@puppet ]# vim /etc/puppet/puppet.conf
[main]
    environmentpath = /etc/puppet/environments

# 查看当前environment的一些默认值
[root@puppet ]# puppet config print all | grep environment
environment_timeout = 0
manifest = /etc/puppet/environments/production/manifests
environment = production
modulepath = /etc/puppet/environments/production/modules:/etc/puppet/modules:/usr/share/puppet/modules
disable_per_environment_manifest = false
environmentpath = /etc/puppet/environments
```

可以看出 environmentpath 已被正确设置。默认的 environment 为 production；默认 production 的 modulepath 为 /etc/puppet/environments/production/modules 加上 puppet 默认的两个；就连 site.pp 也已经开始读 /etc/puppet/environments/production/manifests 下的了。

因此，为了做到彻底隔离，要删除原有的路径下的 site.pp 和 modules。

```
[root@puppet ]# mv /etc/puppet/manifests/site.pp /etc/puppet/environments/production/manifests
[root@puppet ]# mv /etc/puppet/modules/ * /etc/puppet/environments/production/modules

# 再次运行puppet agent -t，以测试迁移的完整性
[root@agent1 ]# puppet agent -t
```

下面说明如何创建一个新的 environment。

```
# 创一个叫test的environment。创建如下结构，modules和site.pp可以拷贝原有production的内容以作初始化用
/etc/puppet/environments/test/
/etc/puppet/environments/test/enviroment.conf
/etc/puppet/environments/test/modules/
/etc/puppet/environments/test/manifests/site.pp

# 修改enviroment.conf
```

```
[root@puppet ]# vim /etc/puppet/environments/test/enviroment.conf
[test]
manifest = /etc/puppet/environments/test/manifests/site.pp
modulepath = /etc/puppet/environments/test/modules
```

虽然上面都是默认值，但是完全可以指定其他不同的路径，不过建议就这样，保持标准。

现在可以测试新环境了，代码如下：

```
# 首先查看新配置是否生效
[root@puppet ]# puppet config print all --environment test | grep environment
environment_timeout = 0
manifest = /etc/puppet/environments/test/manifests
environment = test
modulepath = /etc/puppet/environments/test/modules:/etc/puppet/modules:/usr/
share/puppet/modules
disable_per_environment_manifest = false
environmentpath = /etc/puppet/environments
```

这里在 cmd 里使用了 `--environment`，它可以临时修改本次运行的 environment，还可以通过客户端的 `puppet.conf`，以及上文提到的 `enc` 中的 `environment` 变量来永久修改。

下面安装新的模块到测试环境中并开始测试。

```
# 安装新模块到test环境
[root@puppet ]# puppet module install --environment test puppetlabs/apache
Notice: Preparing to install into /etc/puppet/environments/test/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppet/environments/test/modules
?...? puppetlabs-apache (v1.4.1)
    ...? puppetlabs-concat (v1.2.3)
    ...? puppetlabs-stdlib (v4.6.0)
```

最后，在 `/etc/puppet/environments/test/manifests/site.pp` 中加入 `include apache`，开始测试吧

```
[root@puppet ]# vim /etc/puppet/environments/test/manifests/site.pp
node default {
    include apache
}
```

```
[root@agent1 ]# puppet agent -t --environment test
```

3. 具有版本控制功能的 Puppet 撰写

本节将通过三种模式，来阐述具有版本控制功能的 Puppet 的撰写最佳实践。

(1) 传统模式

传统模式中，多数读者肯定选择直接在 `/etc/puppet/modules` 里进行代码修改，这样做有以下两个风险。

第一，Puppet agent 的 service 运行时，默认是 30 分钟执行一次，很有可能错误地执行了你正在修改的内容。示例如下：

```
[root@puppet ]# vim /etc/puppet/modules/network/templates/resolv.conf.erb
search example.com
nameserver <%- example_nameserver -%>
[root@puppet ]# vim /etc/puppet/manifests/site.pp
node default {
    include network
    $example_nameserver = "10.16.100.10"
}
```

以上 2 步看似很完美，可当你完成第一步的时候，很可能有些 node 已经开始执行了。此时，第二个文件编辑，\$example_nameserver 将会为空，服务器上的 resolv.conf 已经是空值了，即使你后来完成了第二个文件的编辑，但是已经晚了，连修补的机会都没有。因为 Puppet 也依赖 DNS，你还得一台台找哪些错误执行了，而且如果有生产服务依赖于 DNS 去访问其他服务，那么就准备写事故报告了。

不过，别急，还好 Puppet 对付未定义是直接运行报错，不像 bash 那样赋予控制空值，是不是吓出一身冷汗？但下次有可能没那么幸运了，因为有些 Puppet 模块有默认值！

第二，如果团队里有其他人在修改，则很容易产生冲突，甚至会出现各种不可预期的结果。

这基本不需要用例子说明。一个人在改，另外一个人会碰到文件已经打开的错误，聪明点的人会 ps aux，并且询问其他人是不是也在操作，碰到团队里的短板会直接把 vim 的 .swp 文件删掉！当然更容易产生上文中所描述的风险，因为 2 个人有可能在改同一个模块，因此互相影响在所难免。综上所述，传统模式，赶紧摒弃！

（2）进阶模式

在熟悉 Puppet 以后，渐渐可以开始使用 environment 来进行生产环境和测试环境的隔离，并且用 git 来进行代码管理了。因此，Puppet 更改流程变为：

- 1) 在 test environment 下起草 Puppet 代码。
- 2) Puppet 草稿完成后，用 Puppet agent -t --environment test 的方式测试一台机器，或者直接设置业务开发 site 下的所有服务器为 test environment。
- 3) 测试没问题后，推到 git 上。
- 4) 通过 cd 命令切换到 production environment 的 Puppet module 目录，手工触发 git pull origin master，完成到生产环境的更新。

可以看出此流程已经相当具有可控性，风险也降到了极低，但是该模式有如下几个问题。

- ❑ 虽说没有影响 production environment，但也确实影响了 test environment。这也是传统模式中的问题。
- ❑ 需要团队内仔细使用 git。

同样以示例说明，如下：

```
# 用户A在改站点example.com的文件
[root@puppet test]# vim /etc/puppet/environments/test/modules/apache/files/
example.com.conf
[root@puppet test]# vim /etc/puppet/environments/test/modules/apache/files/
example.com_drupal_cron.php
[root@puppet test]# vim /etc/puppet/environments/test/modules/apache/files/
example.com_drupal_cron.d.conf

# 用户B在改站点example2.cn的文件
[root@puppet test]# vim /etc/puppet/environments/test/modules/apache/files/
example2.cn.conf
[root@puppet test]# vim /etc/puppet/environments/test/modules/apache/files/
example2.cn_drupal_cron.php
[root@puppet test]# vim /etc/puppet/environments/test/modules/apache/files/
example2.cn_drupal_cron.d.conf
```

用户 A 首先完成，以为就他一个人在改这个模块，并执行“`git add apache/files/*`”，然后悲剧就发生了，有可能用户 B，刚改好 `example2.cn.conf`，还没来得及改另外两个需要一起修改的文件。好吧，就算用户 A 有这个意识了，他特地用 `git status` 查看了下是否有人更改，因为发现没有，于是很放心地修改这个模块，并执行了 `git add apache/files/*`，但就在用户 A 敲这两条命令的这几秒钟内，用户 B 很凑巧地 `save` 了一个文件，悲剧依然发生。

以上悲剧其实就是为了阐述一个道理，`git add XX/*` 虽然很方便，但是千万别在有任何与他人共享的代码目录里使用。虽然出错概率不高，不过，总有那么一次，要推十几个文件，非要选择 `git add XX/*` 的时候，巧合就发生了，请参考墨菲定律！

综上所述，进阶模式，虽然解决了一部分风险，但却又创建了另外一个风险，鉴于风险较低，依然还是很多团队的首选方式。

（3）高级模式

读者肯定要问，那有没有更好的解决方案呢？答案是有的。下面就来看下笔者实践中正在使用的一种方式。关键字“人人有份，知根知底”。

“人人有份”即每个人都有自己的 `environment`，不仅完美解决在同一目录下冲突的问题，而且也解决潜在的 `git` 错误提交的问题。

“知根知底”即大家都知道团队里都在什么机器上测试什么模块，即使碰到不可预期的问题，也可以一眼看出应该抓谁对齐。

看上去是不是很丰满，其实实现方法也不难，只要完成如下几步。

第一步，统一配置文件 `/etc/puppet/myenv.conf`，代码如下：

```
[root@puppet]# vim /etc/puppet/myenv.conf
laoshi_cang:
    hosts: ['sextest-ntp001', 'textest-ntp002']
    modules: ['ntp', 'motd']
nvsheng_tang:
```

```

    hosts: ['sexdev*']
    modules: ['all']
jieba_xiao:
    hosts: []
    modules: []

```

上面给出了一个简单的 yaml 格式的配置文件，总共涉及 laoshi_cang、nvsheng_tang、jieba_xiao 三位人员。每个人员的 hosts 和 modules 属性的 value 代表了该人员正在测试的 hosts 和模块；hosts 可以是正则表达式，modules 使用 all 代表所有模块。

laoshi_cang 在系统上的用户名应该是 laoshi.cang，但这里用下划线 "_" 替换了点 "."，之所以这样替换，是因为 puppet 的 environment 命名不能接受 "."。它正在占用 hostname 为 sextest-ntp001 和 textest-ntp002 的服务器上，测试 ntp 和 motd 模块。nvsheng_tang 则比较贪婪，不仅拿了 sex 机房（Shanghai example）的所有 dev 机器，而且要测试所有 modules。jieba_xiao 就比较好了，本本分分，没有任何测试任务。

第二步，在 puppet.conf 中使用 tags。示例如下：

```

[root@agent1]# puppet agent -t --tags apache --debug
Debug: Class[Ntp]: Not tagged with apache
Debug: Class[Ntp::Params]: Not tagged with apache
Debug: /Schedule[weekly]: Not tagged with apache
Debug: /Schedule[puppet]: Not tagged with apache

```

可以看出 tags 指定了 puppet 只运行相关模块，会跳过不匹配的模块。当然完全可以通下下面的 puppet.conf 的例子完成持久化配置。

```

[agent]
classfile = $vardir/classes.txt
localconfig = $vardir/localconfig
tags = apache

```

第三步，制作一个可以从 /etc/puppet/myenv.conf 中读取 tags 和 environment 的 enc 脚本。

```

# 这里直接以上文讲过的zabbix作为enc来源进行改写
[root@puppet ~]# cat /usr/local/bin/my_enc.py
#!/usr/bin/python
from pyzabbix import ZabbixAPI
import sys
import re
import yaml
MYENV_CONF = "/etc/puppet/myenv.conf"

class GetParameters(object):
    def __init__(self, host, outcome):
        self.host = host
        self.outcome = outcome

```

```

def get_zabbix(self):
    zapi = ZabbixAPI("http://zabbix.example.com/zabbix")
    zapi.login("admin", "zabbix")
    zbx_get_result = zapi.host.get(output="extend", withInventory="true",
        selectInventory="extend", filter={"host": self.host})
    if zbx_get_result:
        h = zbx_get_result[0]
        for i in h["inventory"]:
            if h["inventory"][i]:
                if i == "tag":
                    self.outcome["environment"] = h["inventory"][i]
                elif re.match("software_app_[abcde]", i):
                    self.outcome["classes"][h["inventory"][i]] = []
                else:
                    self.outcome["parameters"][i] = h["inventory"][i]

def get_myenv(self):
    with open(MYENV_CONF, "rb") as f:
        settings = yaml.load(f)
        f.close()
    if "puppet-master" in self.outcome["classes"]:
        self.outcome["parameters"]["user_env_list"] = {}
        for k in settings:
            ops = re.sub(r'_' , r'.' , k)
            self.outcome["parameters"]["user_env_list"][ops] = k

    for env in settings:
        for h in settings[env]["hosts"]:
            if re.match(r'%s' % (h), self.host):
                self.outcome["environment"] = env
                self.outcome["parameters"]["tags"] = settings[env]["modules"]
                if "all" in self.outcome["parameters"]["tags"]:
                    self.outcome["parameters"]["tags"] = []
                if self.outcome["parameters"]["tags"]:
                    self.outcome["parameters"]["tags"].append("puppet-agent")
                    self.outcome["parameters"]["tags"].append("puppet-master")
                break

def run(self):
    self.get_zabbix()
    self.get_myenv()

    return self.outcome

def main(host):
    default_outcome = {"classes": {"puppet-agent": {}, "puppet-master": {}},
        "parameters": {}, "environment": "test"}
    final_outcome = GetParameters(host, default_outcome).run()
    print yaml.safe_dump(final_outcome, default_flow_style=False)

```

```

if __name__ == "__main__":
    try:
        host = sys.argv[1]
    except:
        print "I need a hostname as sys.argv[1]"
    main(host)

```

本脚本使用了 Python 的 class 概念，具体语法不再展开，有兴趣的读者可以自行学习 Python 相关书籍，由于 Python 是 DevOps 的基本技能，因此建议读者学习，毕竟很多 ops 相关工具都是用 Python 写的。

本脚本关键 function 是 class GetParameters 中的 run(self)，执行了 get_zabbix() 和 get_myenv()，并返回 main 函数 final_outcome。get_myenv() 正确读取 myenv.conf 并赋予相应的 environment 和 parameters 里的 tags 变量，如果该机是 Puppetmaster，则加上 user_env_list 的返回，以便在 Puppetmaster 上部署不同 puppet environment 的设定，输出见下面的代码。puppet_agent 作为默认 classes，方便控制 agent 的 tags 和 environment。



说明 agent 和 tags 会在后文 Puppet agent 模块的代码中使用；master、user_env_list 会在后文 Puppet master 模块的代码中使用。

```

[root@puppet /]# /usr/local/bin/my_enc.py sextest-ntp001
classes:
    ntp: {}
environment: laoshi_cang
parameters:
    name: sextest-ntp001
    tags:
    - ntp
    - motd

[root@puppet /]# /usr/local/bin/my_enc.py puppet
classes:
    puppetmaster: {}
environment: production
parameters:
    name: puppet
    user_env_list:
        laoshi.cang: laoshi_cang
        nvsheng._tang: nvsheng_tang
        jieba._xiao: jieba_xiao

```

第四步，撰写 Puppet agent 的 modules。

init.pp 中的代码如下：

```

[root@puppet /]# vim /etc/puppet/environments/production/modules/puppet-agent/
manifests/init.pp
class puppet-agent {

```

```

package { "puppet":
    ensure => installed,
}
file { ["/etc/puppet/puppet.conf":
    content => template("puppet/puppet.conf.erb"),
    require => Package["puppet"],
]
}
service { "puppet":
    ensure => running,
    hasstatus => true,
    enable => true,
    require => [ Package["puppet"], File["/etc/puppet/puppet.conf"] ],
}
}

```

puppet.conf.erb 中的代码如下:

```

[root@puppet/]# vim /etc/puppet/environments/production/modules/puppet-agent/
templates/puppet.conf.erb
[main]
# The Puppet log directory.
# The default value is '$vardir/log'.
logdir = /var/log/puppet

# Where Puppet PID files are kept.
# The default value is '$vardir/run'.
rundir = /var/run/puppet

# Where SSL certificates are kept.
# The default value is '$conffdir/ssl'.
ssldir = $vardir/ssl

[agent]
# The file in which puppetd stores a list of the classes
# associated with the retrieved configuration. Can be loaded in
# the separate ``puppet`` executable using the ``--loadclasses``
# option.
# The default value is '$conffdir/classes.txt'.
classfile = $vardir/classes.txt

# Where puppetd caches the local configuration. An
# extension indicating the cache format is added automatically.
# The default value is '$conffdir/localconfig'.
localconfig = $vardir/localconfig

tags = <% @tags.each do |each_tag| -%><%= each_tag + ',' -%><% end %>

```

第五步, 撰写 Puppet master 的 modules。以下为示例。

init.pp 中的代码如下:

```

[root@puppet /]# vim /etc/puppet/environments/production/modules/puppet-master/

```

```

manifests/init.pp
class puppet-master {
  package { "epel-release":
    ensure => installed,
  }
  package { ["mod_passenger", "mod_ssl", "httpd"]:
    ensure => installed,
    require => Package["epel-release"],
  }
  package { "puppet-server":
    ensure => installed,
    require => Package["epel-release"],
  }

  file { ["/usr/local/bin/my_enc.py":
    content => template("puppet-master/my_enc.py.erb"),
    mode => "0755",
    owner => "root",
    group => "root",
  ]

  file { ["/etc/puppet/rack/":
    ensure => directory,
    recurse => true,
    owner => "puppet",
    group => "puppet",
    require => Package["puppet-server"],
  ]

  file { ["/etc/puppet/rack/tmp", "/etc/puppet/rack/public"]:
    ensure => directory,
    owner => "puppet",
    group => "puppet",
    recurse => true,
    require => File["/etc/puppet/rack/"],
  ]

  exec { "copy_config_ru":
    command => "cp `rpm -ql puppet | grep config.ru` /etc/puppet/rack/",
    onlyif => "test ! -f /etc/puppet/rack/config.ru",
    path => "/usr/bin:/usr/sbin:/bin",
    require => File["/etc/puppet/rack/"],
  }

  file { ["/etc/httpd/conf.d/puppetmaster.conf":
    content => template("puppet-master/puppetmaster.conf.erb"),
    require => File["/etc/puppet/rack/"],
    notify => Service["httpd"],
  ]

  file { ["/etc/puppet/puppet.conf":

```

```

        content => template("puppet-master/puppet.conf.erb"),
        require => Package["puppet-server"],
        notify => Service["httpd"],
    }

    file { ["/etc/puppet/environments":
        ensure => directory,
        recurse => true,
        owner => "puppet",
        group => "puppet",
        require => Package["puppet-server"],
    ]

    define setup_user_env($user = $title) {
        $user_name = $user_env_list[$user]
        file { ["/etc/puppet/environments/$user_name":
            ensure => directory,
            recurse => true,
            owner => "puppet",
            group => "puppet",
            require => Package["puppet-server"],
        ]

        file { ["/opt/$user", "/opt/$user/modules", "/opt/$user/manifests"]:
            ensure => directory,
            recurse => true,
            owner => "puppet",
            group => "puppet",
            require => File["/etc/puppet/environments/$user_name"],
        ]

        file { ["/etc/puppet/environments/$user_name/environment.conf":
            require => File["/etc/puppet/environments/$user_name"],
            content => inline_template("
manifest = /home/$user/manifests/site.pp
modulepath = /home/$user/modules
"),
            notify => Service["httpd"],
        ]
    }

    $user_env_list_keys = keys($user_env_list)
    setup_user_env { $user_env_list_keys: }

    service { "httpd":
        require => [Package["puppet-server"], File["/etc/puppet/puppet.conf"]],
        ensure => running,
    }
}

```

puppet.conf.erb 中的代码如下:

```
[root@puppet /]# vim /etc/puppet/environments/production/modules/puppet-master/
```



```

    templates/puppet.conf.erb
[main]
    # The Puppet log directory.
    # The default value is '$vardir/log'.
    logdir = /var/log/puppet

    # Where Puppet PID files are kept.
    # The default value is '$vardir/run'.
    rundir = /var/run/puppet

    # Where SSL certificates are kept.
    # The default value is '$confdir/ssl'.
    ssl_dir = $vardir/ssl
    environmentpath = /etc/puppet/environments

[agent]
    # The file in which puppetd stores a list of the classes
    # associated with the retrieved configuration. Can be loaded in
    # the separate ``puppet`` executable using the ``--loadclasses``
    # option.
    # The default value is '$confdir/classes.txt'.
    classfile = $vardir/classes.txt

    # Where puppetd caches the local configuration. An
    # extension indicating the cache format is added automatically.
    # The default value is '$confdir/localconfig'.
    localconfig = $vardir/localconfig

    tags = <% @tags.each do |each_tag| -%><%= each_tag.capitalize + ',' -%><% end %>

[master]
    # Where Puppet looks for template files. Can be list of colon-separated
    # directories.
    # Defaults to '$vardir/templates'
    # templatedir = /var/lib/puppet/templates
    ssl_client_header = SSL_CLIENT_S_DN
    ssl_client_verify_header = SSL_CLIENT_VERIFY
    node_terminus = exec
    external_nodes = /usr/local/bin/my_enc.py

```

puppetmaster.conf.erb 中的代码如下：

```

Listen 8140
<VirtualHost *:8140>
    SSLEngine on
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:RC4+RSA:+HIGH:+MEDIUM:-LOW
    SSLProtocol all -SSLv2
    SSLCertificateFile      /var/lib/puppet/ssl/certs/5eb5ba0b9eb0.example.com.
    pem
    SSLCertificateKeyFile    /var/lib/puppet/ssl/private_keys/5eb5ba0b9eb0.exam-
    ple.com.pem

```

```

SSLCertificateChainFile /var/lib/puppet/ssl/certs/ca.pem
SSLCACertificateFile /var/lib/puppet/ssl/certs/ca.pem
# CRL checking should be enabled; if you have problems with Apache complain-
    ing about the CRL, disable the next line
#SSLCARevocationFile /var/lib/puppet/ssl/ca/ca_crl.pem
SSLVerifyClient optional
SSLVerifyDepth 10
SSLOptions +StdEnvVars

# The following client headers allow the same configuration to work with
    Pound.
RequestHeader set X-SSL-Subject %{SSL_CLIENT_S_DN}e
RequestHeader set X-Client-DN %{SSL_CLIENT_S_DN}e
RequestHeader set X-Client-Verify %{SSL_CLIENT_VERIFY}e

LogFormat "%h %l %u %t \"%r\" %>s %b %D \"%{Referer}i\" \"%{User-Agent}i\""
    puppet
CustomLog /var/log/httpd/puppet.log puppet

# Set this to about 1.5 times the number of CPU cores in your master:
PassengerMaxPoolSize 36

# Recycle master processes after they service 10000 requests
PassengerMaxRequests 10000

# On some systems where disk I/O is expensive, setting this option to a value
    of x means that the above list of filesystem
# checks will be performed at most once every x seconds. Setting it to a
    value of 0 means that no throttling will take place,
# or in other words, that the above list of filesystem checks will be
    performed on every request.
PassengerStatThrottleRate 120

# Since communication with the puppetmaster from puppetd is a long process
    (more than 20 seconds in most cases)
# and will allow for processes to get recycled better
PassengerUseGlobalQueue on

# The additional Passenger features for apache compatibility are not needed
    with Puppet.
PassengerHighPerformance on

PassengerPoolIdleTime 150

RackAutoDetect Off
RailsAutoDetect Off

RackBaseURI /

<IfModule mod_mem_cache.c>
CacheEnable mem /

```

```

CacheDefaultExpire 300
MCacheSize 1024000
MCacheMaxObjectCount 10000
MCacheMinObjectSize 1
MCacheMaxObjectSize 2048000
MCacheRemovalAlgorithm GDSF
CacheIgnoreNoLastMod On
</IfModule>

DocumentRoot /etc/puppet/rack/public
<Directory /etc/puppet/rack>
    Options None
    AllowOverride None
    Order allow,deny
    allow from all
    Options -MultiViews
</Directory>
</VirtualHost>

```

my_enc.py.erb 中的代码如下:

```

#!/usr/bin/python
from pyzabbix import ZabbixAPI
import sys
import re
import yaml
MYENV_CONF = "/etc/puppet/myenv.conf"

class GetParameters(object):
    def __init__(self, host, outcome):
        self.host = host
        self.outcome = outcome

    def get_zabbix(self):
        zapi = ZabbixAPI("http://zabbix.example.com/zabbix")
        zapi.login("admin", "zabbix")
        zbx_get_result = zapi.host.get(output="extend", withInventory="true",
            selectInventory="extend", filter={"host": self.host})
        if zbx_get_result:
            h = zbx_get_result[0]
            for i in h["inventory"]:
                if h["inventory"][i]:
                    if i == "tag":
                        self.outcome["environment"] = h["inventory"][i]
                    elif re.match("software_app_[abcde]", i):
                        self.outcome["classes"][h["inventory"][i]] = []
                    else:
                        self.outcome["parameters"][i] = h["inventory"][i]

    def get_myenv(self):
        with open(MYENV_CONF, "rb") as f:

```

```

        settings = yaml.load(f)
        f.close()
    if "puppet-master" in self.outcome["classes"]:
        self.outcome["parameters"]["user_env_list"] = {}
        for k in settings:
            ops = re.sub(r'_' , r'.' , k)
            self.outcome["parameters"]["user_env_list"][ops] = k

    for env in settings:
        for h in settings[env]["hosts"]:
            if re.match(r'%s' % (h) , self.host):
                self.outcome["environment"] = env
                self.outcome["parameters"]["tags"] = settings[env]["modules"]
                if "all" in self.outcome["parameters"]["tags"]:
                    self.outcome["parameters"]["tags"] = []
                if self.outcome["parameters"]["tags"]:
                    self.outcome["parameters"]["tags"].append("puppet-agent")
                    self.outcome["parameters"]["tags"].append("puppet-master")
                break

    def run(self):
        self.get_zabbix()
        self.get_myenv()

        return self.outcome

def main(host):
    default_outcome = {"classes": {"puppet-agent": {}, "puppet-master": {}},
                       "parameters": {}, "environment": "test"}
    final_outcome = GetParameters(host, default_outcome).run()
    print yaml.safe_dump(final_outcome, default_flow_style=False)

if __name__ == "__main__":
    try:
        host = sys.argv[1]
    except:
        print "I need a hostname as sys.argv[1]"
    main(host)

```

经过漫长的 modules 撰写，现在来享受成果吧！

首先把当前 Puppet 代码推上去，代码如下：

```

[root@puppet modules]# pwd
/etc/puppet/environments/production/modules
[root@puppet modules]# git add *
[root@puppet modules]# git commit
[root@puppet modules]# git push

```

假如开发人员 nvsheng.tang 想测试 ntp 模块。

```

[nvsheng.tang@puppet]$ sudo vim /etc/puppet/myenv.yaml

```

```
nvsheng_tang:
  hosts: ['textest-ntp002']
  modules: ['ntp']
```

nvsheng.tang 是第一次测试，他用 `git clone` 拿到最新代码（第二次可以 `git pull`），代码如下：

```
[nvsheng.tang@puppet]$ git clone ssh://git@sexgitserver/home/git/puppetmodule/
modules
[nvsheng.tang@puppet]$ cd modules
[nvsheng.tang@puppet modules]$ ll
drwxr-xr-x 8 puppet puppet 4096 Apr 28 18:10 apache
drwxr-xr-x 7 puppet puppet 4096 Jun  2 19:58 concat
drwxr-xr-x 7 puppet puppet 4096 May 27 11:39 ntp
drwxr-xr-x 3 puppet puppet 4096 Jul  1 14:57 puppet-agent
drwxr-xr-x 4 puppet puppet 4096 Jul  3 13:23 puppet-master
drwxr-xr-x 6 puppet puppet 4096 Apr 15 20:11 stdlib
```

他修改了 `ntp/templates/ntp.conf.erb`，如下：

```
[nvsheng.tang@puppet modules]$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   ntp/templates/ntp.conf.erb
#
no changes added to commit (use "git add" and/or "git commit -a")
```

然后就直接在 `textest-ntp002` 机器上运行，如下：

```
[root@textest-ntp002 ]# puppet agent -t
Warning: Local environment: "production" doesn't match server specified node
environment "nvsheng_tang", switching agent to "nvsheng_tang".
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Info: Caching catalog for 5eb5ba0b9eb0.example.com
Info: Applying configuration version '1435936930'
```

他看见机器已经被自己接管，于是开始心满意足的测试。完成测试后推到 `gitserver`。

```
[nvsheng.tang@puppet modules]$ git add ntp/templates/ntp.conf.erb
[nvsheng.tang@puppet modules]$ git commit
[nvsheng.tang@puppet modules]$ git push
```

nvsheng.tang 最终推到所有生产服务上。

```
[root@puppet modules]# pwd
/etc/puppet/environments/production/modules
[root@puppet modules]# git pull
```

事后，他非常得体地清理了现场。

```
[nvsheng.tang@puppet]$ sudo vim /etc/puppet/myenv.yaml
nvsheng_tang:
  hosts: []
  modules: []
```

至此，高级模式最终完成，虽然历经艰辛，不过相信读者的收获也不小，以后测试会更舒服，投入到生产环境也更放心。

10.2.4 确保你的代码不是留给别人的坑

写了 Puppet 代码那么多年，看到现在所有生产服务器的角色都可以做到扩展后开机即用，笔者觉得对于团队来说，也是一种值得骄傲的成就，成就感丝毫不亚于性能调优、私有云等看上去高大上的项目。如果说性能调优是经验和理论知识的体现，私有云是对于潮流技术的快速学习能力肯定的话，那么 Puppet 撰写绝对是一个工程师思维缜密和做事态度的考验。对于一个团队来说，能够进行性能调优和私有云优秀人才是不可或缺的，但是也需要更多 Puppet 写得漂亮的人，因为这能带动团队良好的氛围和做事方式，线上业务也就会更稳定。

接下去，回到 Puppet 代码，列举一下在这上面常见的坑。

1. all git

第一个也是最基本的军规，任何代码请做好代码控制。代码控制的好处不仅是可以追溯历史，还可以做到备份和快速回滚，更重要的是，是大家对于彼此的一种信任，信任 git 的 puppet code。

如果非要在 /etc/puppet/environments/production/modules/ 下面直接 dirty 更改，而不是按照流程测完 git push，再到 /etc/puppet/environments/production/modules/ git pull 的话，那不是偷懒，是对于别人的不尊重，是对信任的一种践踏，这个坑在笔者团队中出现的次数不过 3 次，不过每次出现，“坑神”都会被骂得很惨。

2. 代码中对于各种 layer 进行 if else

第二个是非常常见的坑，挖这个坑的理由通常是懒得在 hiera 或者 enc 中顶一个变量，结果是所有 Puppet 模块中充斥着这种代码，用 hiera 或者 enc 集中管理的失去意义，原来遵守规则的人都变得随意，团队氛围开始变味。示例如下：

```
if $datacentre in ["sex", "tex"] and !($hostname in ["sexntp01", "texntp02"]) {
  service { "iptables":
    ensure => stopped,
  }
}
else {
  service { "iptables":
```

```

        ensure => running,
    }
}

```

hostname 级别的 dirty code，那是更加肆意妄为的表现。当集群中有一台配置出问题的时候，别人不得不用各种 grep 找所有模块中是否有 datacentre、project、hostname 等级别的坑，而不是通过 enc/hiera 中一个 is_iptables_enable 变量直接比较与另外一台的区别。

3. 各种写死

第三个是一种 Puppet 新手常见的问题，就是以解决问题为目的，而不考虑可持续发展的代码，示例如下：

```

<% if datacentre== "sex" -%>
echo '##### fix route####'
IPADDR=`ip addr | grep "10\[0-9\]*\." | awk '{ print $2 }'|cut -d"." -f3`
if [ $IPADDR -eq "43" ]
then
    /sbin/ip ro add 10.123.41.0/25 via 10.123.43.1
    /sbin/ip ro add 172.16.0.0/12 via 10.123.43.1
    /sbin/ip ro add 10.16.0.0/14 via 10.123.43.254
fi
if [ $IPADDR -eq "44" ]
then
    /sbin/ip ro add 10.123.41.0/25 via 10.123.44.1
    /sbin/ip ro add 172.16.0.0/12 via 10.123.44.1
    /sbin/ip ro add 10.16.0.0/14 via 10.123.47.254
fi
<% end -%>

```

在上述代码中，第一，用了 datacentre 级别的 if else。第二，这些不同网段的静态路由，肯定会在其他地方用到，如果写死在模块里，其他模块再写一遍的话，很容易出现不一致的现象。第三，不中央管理的写死，极易出现变更的时候，漏到这个代码，比如要改了一个网段的默认路由，这个模块就会是一个深坑。

4. 不顾变化的临时代码

比如，对一个开源的监控软件 Zabbix，进行一次个性化代码更改，如下：

```

file { ["/usr/share/zabbix/include/defines.inc.php":
    ensure => present,
    mode => 644,
    owner => 'apache',
    group => 'root',
    require => Package["zabbix-web"],
    source => "puppet:///modules/zabbix-server/defines.inc.php",
}

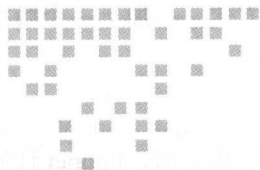
```

这样的代码，就是以后一个“神坑”，升级 Zabbix rpm 的人，非常有可能会忘记这个

改动，然后监控系统就出现了不可预期的问题。好的方式，是加上一个版本的限制，比如只有这个版本的 zabbix-web 才执行这段 Puppet 代码。

5. 代码 review

所谓智者千虑必有一失，一个人很容易走到思维定势里面，Puppet 代码也是一样，写好之后感觉非常好，测了 2 台机器都没问题，兴冲冲地推到生产的时候，基本就要写事故报告的节奏。因此，代码 review 是相当重要的，这里推荐 2 个开源工具：Gerrit 和 GitLab，具体如何搭建，这里不再展开，笔者项目正在用 Gerrit，这是老牌的工具，但笔者更喜欢 GitLab，很有 GitHub 的范。



Chapter 11 第 11 章

CMDB 配置中心管理

11.1 什么是 DCIM

DCIM 全称为 Data Center Infrastructure Management，即数据中心基础设施管理。具体展开概念前，笔者想问几个问题，请问在贵公司：

- ☐ 有多少个 IDC？
- ☐ IDC 内有多少个房间？
- ☐ 有多少个机柜？
- ☐ 有多少个交换机？
- ☐ 有多少台机器？
- ☐ 机器与机柜的位置关系是什么？
- ☐ 机器与交换机端口的连线关系是什么？
- ☐ 机器的角色是什么？
- ☐ 一个机柜交换机的使用和剩余是多少？
- ☐ 一个机柜电源插口的使用和剩余是多少？
- ☐ 一个机柜的电力使用和剩余是多少？
- ☐ 一个机柜的空间使用和剩余是多少？
- ☐ 房间制冷是否正常，是否有局部过热或者个别机器过热现象？

也许很多人会回答：“我们有很多 excel 表格，应该还没过期吧！关于电力和房间我们依赖 IDC 供应商发报告通知我们。”如果说后者还可以忍受的话，那用 excel 表格简直是无法忍受（当然如果目前机器规模较少，只有几十台那还勉强可以接受，又或者读者在云

端……)，为什么这么说呢？来看下以下几个场景。

- ❑ 项目快速增长，需要购置 50 台机器，但需要评估有哪些 IDC 相关资源需要扩容。
- ❑ 项目运行两年多了，但有些设备或者组件会过保，需要续保或更新，如何找出它们？
- ❑ 项目需要，需要迁移一些机器到其他机房，如何制定一个详细的计划，可以使实施小组快速准确。

相信有用过 excel 表格的读者，这时候肯定对手头的数据一百个不相信，于是一个费时费力又无聊的审计任务便开始了，一次两次三次，不仅严重拖累了团队效率，而且即使审计完成，也有很高的出错机率，因为没人能保证一个耗时的工作会没有人出错，尤其还是一个无聊的工作。

因此在这种因素下，就诞生了 DCIM 的概念和相关的工具，对于一个决策者来说，没有什么比一个可视化的、可信任的、具有相互关系的 IDC 数据更具有可操作性了，丢掉手头的 excel 表格吧，让我们像 DevOps 在 server 上管理 App 一样地智能化 DCIM！

11.2 什么是 CMDB

CMDB 全称为 Configuration management database，说白了就是配置管理的一个中心，每个配置称为一个 CI (Configuration Item)。CI 可以有多种分类。上节提到 DCIM 里面的相关的信息都可以转化成一个 CI 存到 CMDB 里，以供使用。CMDB 最早是出现在 IT 管理领域的，是作为 ITIL 必要的基础而存在的，因此为了抽象现实情况到管理流程，往往会定义精细度非常高的 CI，并定义各个 CI 的复杂关系映射，从而实现 ITIL 流程化管理。本书由于篇幅有限，且侧重点不同，并不会讲解 ITIL 的内容，而是专注于 CMDB 和互联网项目 ops 的相关内容。

再回到上节提到的 DCIM，读者肯定会问，DCIM 好像和 CMDB 有重复的内容嘛！确实，两者是会有重合，但却是相辅相成的，DCIM 侧重于机房的基础设施管理，用于展现机房的现实情况和给 IDC 部门提供决策需要信息，而 CMDB 更侧重于提供业务逻辑的信息，比如该机柜有哪些机器，这些机器属于哪些项目、哪个环境、哪个角色组，甚至是该机器在负载均衡器上应该有的权重等。诚然，一个管理有序的大型互联网项目与一个井井有条的 DCIM 系统密不可分，它不仅需要有自动发现的功能，还需要有非常良好的 UI 来展现 IDC 基础设施的拓扑。而 UI 一直都是开源软件的弱项，在 DCIM 领域可谓是商业软件遍地开花，甚至有些软件做到了 3D 视图和全景俯瞰，还和 IDC 的供应商有合作，做到了温度、电力、网络状况等全覆盖，可谓是面面俱到，不得不叹服其强大！当然越好的软件，价格也越贵，这里就不再介绍了。因此，我们也不考虑 DCIM 的因素，后文直接切入 CMDB。

com/wp-content/uploads/2015/01/Gartner-2014-Magic-Quadrant-for-DCIM.pdf, 可以看出竞争也颇为激烈, 如果有需要商业软件的读者, 可以自行研究。笔者公司在用 device42, 它是一个非常优雅的软件, 虽然 UI 没那么酷炫, 也没有详尽的温度、电力、网络报告, 但 2D 的拓扑图已经够用, 笔者所在团队看中了其 CMDB 的亮点, 而且价格算是商业软件里非常公道的, 1000 台的价格才 5000 美元 / 年 (其他酷炫的商业软件基本都是一两万美元 / 年), 当然这里没有做广告的意思, 后文关于 CMDB 的讲解, 也会选取开源软件作为例子。

11.3 运维为什么需要 CMDB

前文已提到 CMDB 对于 ITIL 的意义, 可是除此以外, 运维为什么还需要 CMDB 呢? 下面将展开说明。

11.3.1 整合信息

所谓整合信息, 是把零散的重要信息整合到一块, 使团队内部可以方便地共享信息而不需要权限登录去翻箱倒柜地找寻信息。比如, 网络运维可以把 VLAN 信息同步到 CMDB, 而不需要给其他组的同学开放权限, 系统运维可以把 Linux 的系统资源信息同步到 CMDB, 同样不需要权限赋予和增重其他组的学习成本。

11.3.2 关系映射

关于配置项之间的关系, 涉及从属、依赖、并行、互斥等各种关系, 有些显而易见, 有些对于不熟悉的读者来说有可能是一个深坑, 很容易造成人为错误。比如:

- ❑ 在从属关系中, 对于 VLAN 和 IP, IP 中的 netmask 只能继承 VLAN 的 netmask。
- ❑ 在依赖关系中, dev 环境需要一个 Python 模块的包, 这个是一个隐藏非常深的坑, 开发人员往往只跟一两个运维人员说过, 其他运维人员绝对会一个个往坑里跳, 如果在 CMDB 中定义 2 个配置项, 并定义关系, 事实上就可以有效地填平这个坑。
- ❑ 在并行关系中, dev 环境和 staging 环境的代号, 这个也可以是配置项, 而且是很明显的并行关系。
- ❑ 在互斥关系中, 在 CentOS 6 中装 Puppet 2.7+ 默认的 Ruby 1.8.5 会产生内存溢出问题, 需要安装 Ruby 1.8.7 或以上的安装包, 这个也是一个运维常见的问题。而 CMDB 会让这种关系展现出来, 即是一个知识库, 也可作为自动安装 Puppet 的来源方法。

11.3.3 防止配置偏差

读者知道, 运维中另一头疼的问题, 是手头的资料和线上环境不符, 比如明明是 dev

的机器，由于紧急情况被挪到 production 环境应急，如果是做容量规划还好，但如果是做 dev 部署，又碰到比较马虎的“攻城狮”，很容易就看也不看，直接把那台机器当 dev 环境了。所以 CMDB 里面另外一个重要的概念就是自动发现，它必须自动发现该机器的所属环境配置已经变为 production，这样在批量操作 dev 环境的时候不会选取到该台机器。

11.3.4 自动化

自动化的传统意义这里就不展开了，无外乎效率和降低人为出错机率这两点。在 CMDB 中，自动化是核心，因为这是 DevOps 存在的意义之一，一个好的 CMDB 如果没有好的自动化工具使用它，是空有躯壳没有灵魂的。而自动化也是在使用 CMDB 的过程中，最耗时的一环。本章的主要内容也会涉及很多自动化的思路和代码示例。

11.3.5 中央管理

这里说的中央管理主要指可以在 CMDB 一个地方增删改配置，就直接作用于服务器上的配置，这和上一点自动化密不可分的。中央管理的好处不言而喻，这是运维上班喝咖啡看报纸的必要条件。

总而言之，对于一个大型项目而言，在一个好的运维团队中，CMDB 是不可或缺的。

11.4 如何选择适合的 CMDB

11.4.1 每个项目都会遇到的那些任务

如果通过一个实际任务来进行需求分析，那么分析的结果就可以水到渠成地告诉我们在选择 CMDB 的时候，需要考虑哪些因素。现在，一起来看一下这个实际的任务，即从服务器上架、分配角色，到安装操作系统和部署软件的整个过程。

这个任务应该是每个项目必须面对的问题，尤其是数量多的情况下。大致可以分为如下几步：

- 1) ops 和 dev 一起评估所需新机器数量。
- 2) 从电力、网络设备容量、机柜空闲率等方面来评估机房容量。
- 3) 制定扩容方案，包括新机器放置位置、交换机连线和角色分配等。
- 4) 服务器到位，根据方案实施上架和连线。
- 5) 检查硬件和网络，并安装操作系统。
- 6) 根据不同角色分配、部署不同的业务软件。
- 7) 测试完成后上线，接入各系统（如 LB、监控等）。

这 7 大步骤应该是读者在各自项目中都会遇到的基本流程。当然最原始的方法应该都是熟知的，其中痛苦的领悟，想必人人都有。因此，每个项目都会有人或或多或少地进行着

自动化的尝试，或者已经开始尝试使用 CMDB。下面就结合 CMDB 和自动化，把每一步骤中可以优化的过程整理如下，抛砖引玉，如果读者有更优或更适用于自己项目的方案，欢迎交流。

第一步，ops 和 dev 一起评估所需新机器数量。

使用 CMDB 查看已有相同角色机器的数量以及当前空闲机器的数量，节省规划和审计时间，这里在乎的是 CMDB 有整洁的 UI 或者有简单的 API 可以查看机器的角色和状态。

第二步，从电力、网络设备容量、机柜空闲率等方面评估机房容量。

使用 CMDB 查看机柜空闲和交换机空闲端口信息，加快审计和规划速度，这里在乎的是 CMDB 的 DCIM 信息有足够的颗粒度，如交换机端口占有信息。

第三步，制定扩容方案，包括新机器放置位置、交换机连线和角色分配等。

使用 CMDB 信息制定准确上线的操作计划，比如新机器放在哪个位置，哪个网卡，接交换机哪个端口，最好可以直观地让现场人员看到用户的计划。

第四步，服务器到位，根据方案实施上架和连线。

到达这步之后，后面的自动化工作也随之多了起来，在服务器到位并按照方案上架连线后，最重要的事情便是将部分基础信息录入 CMDB，这里部分基础信息指的是服务器硬件基础信息。包括：

- ❑ 类型，这边指的类型可以是内部对于一种型号的定义，比如 DB_SPEC_1，型号是 DELL R720（子信息是包括 2 个 E5-2600 的 CPU，32GB 的内存，PERC H810 的 RAID 卡配 SAS 15K*4 做 RAID 10）。
- ❑ eth0/eth1/drac0 的 mac 地址。
- ❑ 服务器标识，可以用主板的 serial number，推荐用主板上可自定义的 Asset Tag，用自己项目独有的命名规范来进行服务器唯一标识。
- ❑ 服务器的 rack 位置。
- ❑ 服务器的连线端口信息。

以上五个任务要录入 CMDB 并实现自动化的话，需要 CMDB 的自动发现功能做以下几件事情：

- ❑ 利用 ipmi 工具自动发现类型、eth0/eth1/drac0 的 mac 地址、服务器标识等，并录入。
- ❑ 从 CMDB 中查到所有接入层交换机，并访问并通过端口 arp 信息，获得连线信息，并录入。

总结，此时 CMDB 中应该有新上架机器的类型，eth0/eth1/drac0 的 mac、rack 位置和连线端口信息（由于 rack 和连线信息的父配置项是 dc，所以 host 也继承了 dc 的配置项），以及刚设置好的 Asset Tag。这些工作基本是靠一个自动发现结合自定义的 ipmi 工具实现并录入，另外需要设置一个服务器状态 status，比如为 racked，已上架，目的是追踪服务器的状态以便后续工作展开。当然，在多项目的情况下，CMDB 中可以多加一个所属业务（project）的字段，并且设置关系映射，连接这批 host 的业务配置项。

第五步，检查硬件和网络，并安装操作系统。

在把服务器从 IDC 部门移交给系统部门之前，最重要的一环便是检查。比如网络连通性和硬件可用性，这项检查是相当重要的，如果把不正常的服务器移交给系统部门人工检查，不仅仅会影响服务质量，而且还会增加两个部门的交流成本，严重影响交付给业务的进度。那么怎么做到这个自动化呢？基本上，IDC 部门自制的 mini ISO 就可以解决这个问题，具体实现步骤如下：

- 1) 从 CMDB 中设置状态 `racked` 的机器，并设置 `eth0` 为 `pxe` 启动，重启这些机器。
- 2) 通过 `pxe` 引导 mini ISO，推荐以一个跑在内存里的小 image 作为该 ISO。
- 3) mini ISO 启动后，先检查网络，确认服务器各网卡的连通性，以及与交换机协商后端口速度是否正常。
- 4) 然后它会用自带的工具进行硬件检查，比如内存、磁盘、CPU、网卡。
- 5) 完成网络与硬件的检查后，mini ISO 会判断结果，并反馈给 CMDB，如果全部正常，则设置 CMDB 中的 `status` 字段为已检测 `tested`，否则设为 `tested_fail`，并在 CMDB 其中一个字段（比如 `tested_fail_reason`）填写失败原因。
- 6) 当然，任何程序都有失败的时候，检测程序也不例外，运行环境因素和自身 bug，都会造成检测结果最终没有写回 CMDB，由于此时硬件本身就是未验证的状态，因此，可以在第 1 步设置一个 `timeout`，比如 30 分钟内依然无法完成 `test`，便自动设置 `status` 为 `tested_fail`，`tested_fail_reason` 填为 `timeout`。
- 7) 查出状态为 `tested_fail` 的服务器，排错后重新从第 1 步开始。

可以看出，做这个 mini ISO 是有一定工作量和难度的，需要网络、硬件、`pxe`、`devops` 等全面的知识，建议 IDC 部门和系统部门通力合作完成。

最后便是正式移交给系统部门了，安装正式的基础操作系统，安装需要的各种 agent，比如，`Puppet`、监控、日志、`ssh`、`rundeck` 等，或者 `DNS`、`ntp` 之类的基础服务。可以默认集成 `Puppet agent` 到正式的 image 中，并把后续初始化工作交给 `Puppet`，`Puppet` 可以从 CMDB 中读取 `host` 的已有配置项进行个性化设置。比如根据 `dc` 改 `log` 和监控的 `server` 指向；根据硬件类型配置不同 `raid` 卡监控。通过 `firstboot` 脚本发现 `Puppet` 初始化完成后，设置 `status` 为 `system_ready`，或如果 `system_ready_fail` 初始化失败则由系统组进行排错，直至 `system_ready`。

综上所述，CMDB 中虽然只有 `status` 在变化，但是其实在初始化系统的时候，就可以根据 `dc level` 在 CMDB 上设置一些子配置项，比如该 `dc` 应该有监控 `server`、`log server`、`dns server` 等，以方便 `Puppet` 初次运行的自动化，也实现了通过 CMDB 进行真正的中央管理。

第六步，根据不同角色分配，部署不同的业务软件。

这时，系统部门真正可以移交已初始化系统的服务器给业务部门，业务部门可以根据之前的需求分析，在 CMDB 上用 UI 的批量修改功能进行角色分配，当然也可以导出所有 `system_ready` 的机器 `csv`，并用 `excel` 表格规划角色，再用 CMDB 自带功能，或者写脚本调

用 API 进行 csv 导入 CMDB 完成修改。

之前的需求分析可以包括如下信息：role（角色，apache/mysql/redis）、env（环境，dev/staging/prod）、app（应用，商城/门户/后台）等这些上层较为明显的配置项。

当然，光靠这些信息一般无法完成完整的部署，因为在示例中，这个 host 只是一个商城的 Apache 和 PHP server，还需要有代码的版本号，对应的 DB/redis server，建议这种 level 的配置项以 env 和 app 子项的方式配置，并通过继承传给相应的机器。

最后，最下面一层 level 便是 host level 的个性化配置，比如有些 host 就是不想遵循 env+app level 的配置，希望 CMDB 中有个位置可以覆盖原先默认配置项。比如新的一批机器硬件条件比较好，需要在负载均衡器中有更高的权重等。

从这一步来看，CMDB 中具有结构化从属关系的配置项，更能映射出项目的现实情况，再加上一些 host level 的项目可以更好地完成个性化的设置和管理，最后具有强大 API 和 UI 批量操作页面，可以让业务规划更从容。

第七步，测试完成后，上线，接入各系统（如 LB、监控等）。

当然在最后一步里，便是一些无法嵌入的手工步骤，比如：

1) 数据库同步，导入。

2) 更改当前 env+app level 配置项，让老机器添加新节点的应用连接，如让 app 使用新的 db 分片，或者让 HAProxy 添加新的 redis 节点。

3) 运行业务相关自动测试脚本，验证新进节点的业务功能的正确性。

最后，设置 status 为 live，完成上线，Puppet 之类的工具会检查 CMDB 中的状态，完成相关周边系统的更改，比如在监控系统中，切换该机器的监控级别为 24×7 小时，在日志系统中，设置相应的 filter，分析出该业务专属的 log 类型，又比如更改 ssh 的配置文件，限制访问权限，开发从此只有只读权限，开启审计，等等。

到这里，一个完整的通过 CMDB 加上一些自动化工具实现的部署上线任务就算完成了。可以看出，在每一个步骤都会涉及 CMDB，比如：

❑ 获取信息。

❑ 中央 UI 的配置管理。

❑ 通过 CMDB API，开发适用于本项目自动化工具。

因此，可以得出以下选择 CMDB 的基本考虑因素：

❑ CMDB 需要提供自动发现的功能，或者 API 方便自定义自动发现功能。

❑ CMDB 需要有 DCIM 的相关管理功能，比如机架信息、交换机端口信息。

❑ CMDB 需要有硬件基础信息，并能通过从属映射关系方便的管理。

❑ CMDB 需要有批量操作的 UI，或者 API 方便自定义批量更改操作。

❑ CMDB 需要有结构化配置项和定义配置项之间关系的能力，最低要求是从属关系，并有继承和覆盖的功能。

❑ CMDB 需要有良好的初始化配置项结构，以较少初期搭建成本，而且还有个性化配

置项的功能，以方便根据不同业务进行定制。

- ❑ CMDB 需要有强大的 API 以便其他工具脚本进行调用，需要满足性能好、友好、全面等特点。
- ❑ CMDB 需要有比较好的 UI 展现，方便归类、过滤、统计，以完成审计、查找信息、容量规划等多种业务上的常见需求。

可以看出，一个好的 CMDB 要求其实还是很高的，不是一个简单的开源软件可以达到的，用户能做的就是，进行一些取舍，并且寻求对舍去功能的替代解决方法，接下来的一节，会剖析部分开源软件。

11.4.2 选择开源的 CMDB

上文中的业务场景是一个完整的 dc 建设，包括服务器上线的生命周期，对于一个 CMDB 软件来说，具有非常大的挑战，不仅需要成熟的后端架构，有着详尽的 API 和满足高并发场景的能力，而且需要有友好的 UI，有清晰展现和批量操作的能力。这些因素导致能满足此类需求的商业软件价格令人咋舌，笔者项目选择比较便宜且实用的 device42，一年也要好几万美元，当然物有所值是肯定的，它可以省下相当多的人力成本。因此，想在开源解决方案中找到一款类似的软件，几乎不可能，因为维护这套软件的人力成本不是一个松散的开源组织可以支持的。特别是 UI 方面，一直是开源软件的软肋，而 DCIM 功能又是一个考验 UI 开发功底的特性。但是，CMDB 是运维自动化的灵魂，对于大中型项目来说是不可或缺的环节，甚至有些大型公司开发了属于自己的 CMDB，所以，笔者还是希望给读者做出一个由多个开源软件构成的搭积木方式的解决方案，来阐述实施 CMDB 的过程，以及一些鲜活的例子，让读者对于 CMDB 的使用方法更加直观。

接下来，将会列出一些重要的组成部分，根据每个部分选取开源解决方案，并给出笔者调研过软件的评估结果。

1. DCIM 功能

表 11-1 给出了实现 DCIM 功能的一些开源方案，其中考虑了需要有 dc、room、rack、patch panel 等信息，还需要用 UI 直观展示，并且可以方便地批量操作等因素。

表 11-1 比较具有 DCIM 功能的开源方案

项目名称	优势	劣势	评估结果
rackmonkey	小巧，易于开始，用过的评价都很高	不再开发，没有源生 API	放弃
racktable	更新勤快，而且是 PHP 语方的，项目 8 年了，坚毅	作者是老成牌，UI 和 community 都做得一般，8 年还在 0.20.10，没有任何商业相关，真不敢想象项目的将来，估计会和 rackmonkey 一样	不推荐

(续)

项目名称	优势	劣势	评估结果
ralph	UI 比较漂亮，尤其在 rack 管理这块。而且有服务器基本信息管理，兼具自定义 cmdb 的 ci 和关系的功能，开发比较勤快。看得出是一家公司的内部项目，后进行开源的解决方案，开放 jira agile board，并且对于 github 提出的 issue 响应及时	项目依然较新，笔者曾经天真的以为这是 All in one 的解决方案，花了大力气学习其 API 和通读文档，可惜最后开始实行的时候，发现 bug 相当多，文档还是较为简单，API 也不完善，还有非常重的内部项目的影子，很多写死的地方	待观察，由于项目较新，又有一家公司在背后支持，而且他们也发现了目前的问题，正在开发全新的版本，可以保持关注
opendcim	ui 虽然不漂亮，但实用且人性化，目前做的算很好的 oss，开发也比较勤快	API 在开发，感觉是一个非常专注于 dcim 的工具，笔者原以为 ralph 可以胜任，可惜 bug 实在太多。而 opendcim 给了我相当多的惊喜，稳定和几乎无发现 bug，外加 PHP+MySQL 实在方便维护，用户体验非常不错	可以作为 dcim 的优选解决方案

2. 服务器基本信息

表 11-2 列出了在服务器的基本信息时可选择的开源方案，这里的基本信息包括型号（不仅是服务器厂家的型号，还包括当前内存、CPU、硬盘）、网络配置信息（ip/mac/gw）、所属项目（游戏 / 商城 / 官网）、所属环境（prod/test）、角色（web/mysql/mail）等。

表 11-2 比较具有服务器基本信息的开源方案

项目名称	优势	劣势	评估结果
itop	产品线从 opensource 到 commercial，够成熟，也是在国内认知度比较高的产品。由于对于 IT 领域和 itil 做的相当全面，不少非互联网企业都是用它来管理 inventory 的。技术方面，有良好的文档、API，sourceforge 上排名又高，应该是不错的	太偏 IT 了，有可能花时间可以定制，但是代价会不小	不推荐
collins	用的都是最时髦的技术，从 UI、API 分格和 Docker 化都能看出，属性也很全，更新也很勤快	从 github 的历史来看，作者在 2012 发力，开发了 2 年，贡献了 90% 的代码，可惜的是，不知道什么原因，作者失去了动力，虽然有另外一个贡献者陆续写了半年多，但是最近 2 年也就不了了之了	不推荐
ralph	除了具有 itop 和 collins 的所有优势以外，在服务器基本信息这块，更是 ralph 非常出彩的地方，它是完全按照互联网公司的思路去设计的，比如默认有“所属项目”“所属团队”“所属环境”等非常实用的属性，并且还有对于一个项目的基础设施的花费开销的管理，例如 dc 租赁费用、机器采购价格、续保时间提醒等功能	除了上文表 11-1 中描述的劣势，还是需要吐槽一下文档缺乏和很重的内部项目痕迹这两点，真的让笔者走了不少弯路	和上文表 11-1 中的评估结果一样，值得期待，毕竟是新项目，需要点时间。笔者在实践过程中，发现有一些提交的严重 bug 还没在交稿前完成，不得不暂且搁置这个解决方案

(续)

项目名称	优势	劣势	评估结果
Zabbix	一套非常著名的老牌监控系统, 有 inventory 的功能, 而且 API 相当可靠和完善。此外, 还可以根据监控项取数据作为 inventory 的动态数据, 比如某个服务的当前版本。在服务器基本信息这部分, 堪称完美	学习成本有些大, 虽然笔者接触 Zabbix 已有 7 年, 从 1.6 版本时代到如今 3.0 版本, 笔者见证了它的不断演变。接触到的新人同事大部分都感觉上手不易, 不过深入学习后, 基本都感叹其功能强大	极力推荐, 如果本次比较只有“服务器基本信息”这一标准的话
opendcim	除了上文已经描述的优势, opendcim 的服务器基本信息也算够用, 能自定义算是对其少量的预置信息项的一个完美补充	相比 Zabbix, 差了动态效果, 需要自己写工具调用 api 做到自动更新	除了 Zabbix, 第一推荐

3. 配置参数信息

要获取更加细致的配置信息, 比如这个项目的 Puppet master 的 IP、Zabbix server 的 IP、Logging server 的 IP 等, 还有商城业务的数据库配置信息, prod 环境的 storage 地址和读写 I/O 限制。像这些颗粒度非常小的配置管理是大型项目的难点所在, 虽然可以通过写死在每个 Puppet 模块内来实现, 但是, 这造成的问题就是, 后期的维护成本巨大, 且容易忽略某些特殊配置, 碰上各种坑, 因此, 一个具有配置参数信息的集中化工具, 也是需求的功能之一。表 11-3 同样给出了相应的开源项目供参考。

表 11-3 比较具有配置参数信息的开源方案

项目名称	优势	劣势	评估结果
Hirea	小巧, 易于开始, 用过评价都很高	Puppet 官方对 Hirea 的支持, 笔者一直觉得不够到位, 没有原生 API, 目前有在开发一插件支持, 但进度实在太慢, 大概 enterprise 的 killing feature 吧, 对社区版支持力度偏小	放弃
Cmdbuild	Cmdbuild 是一家意大利公司开发的, 优势很意大利, UI 比较优雅, 属于 IT 人员一看就很喜欢的层级式	劣势也很意大利, 很随意, 或者说是懒, 至今文档也不齐, 文档结构也很松散, 挖很深才能看到过期的 API 文档。而且真的需要很大力气去定制, 只提供最小的“螺丝”和“配件”, 绝不会浪费力气预设一些常用的“模块”, 入门非常难。所以他们还没迁移到 github 也不难理解了, 生活那么美好, 哪有那么多精力和社区进行交流呀	不推荐
Zabbix	有 API	不支持 host level 以外的配置参数信息录入, 唯一可以利用的小技巧是, host 链接 template, 用 template level 上的 macro 来实现非 host level 的配置参数。这就需要写脚本跑一个 cron, 逻辑是这个 host 属于哪个 project, 应该有相应的 template 与之对应, link 起来。目前可以利用的就是 hostgroup, 只要 template 和 host 属于同一个 hostgroup, 那么就进行 link	可以看出来这种方式, 逻辑比较复杂, 但至少可以实现
OpenDCIM	有 API	和 Zabbix 一样, 不支持 host level 以外的配置参数信息录入, 唯一可以利用的小技巧是, 创建一个新的 virtual_host, link 一个新的 device template, 配置这个 template 的自定义属性。关联起来的方式是用 environment 的名字作为关联条件, 只要 virtual_host 的名字以 environment 作为前缀的, 则关联	可以看出这种方式, 也较为复杂, 但至少可以实现

根据上文针对各功能解决方案的评估，可以得出以下 3 个组合。

- ❑ **Ralph only**：笔者当初的第一选择，结果被迫走了很多弯路，目前搁置，日后可以继续观察。
- ❑ **openDCIM+Zabbix**：该组合选取了各功能模块最容易实现的解决方案，实现起来应该是最快的。缺点是要维护两个系统的关系，同步以及界限划分，比如，哪些属性是绑定在 openDCIM 上的，哪些属性是绑定在 Zabbix 上的，这些信息系统维护人员需要用户对齐。
- ❑ **openDCIM Only**：好处是单一系统，管理方便，又可以定制，创造出无限的可能性。缺点是相比 Zabbix，项目热度还差点，在实现非 host level 的配置参数录入功能方面，Zabbix 要更好。

说实话，笔者在走过 ralph 的弯路后，已经不敢在交稿前研究不太热门的开源项目了，因此 openDCIM only 和 openDCIM+Zabbix 两个方案就成为了笔者的首选，又鉴于本章目的是解决上文提到的“每个项目都会遇到的那些任务”，故而不考虑把 CMDB 架构弄得太复杂，以致偏离了主题。最终，笔者选择 openDCIM only 作为实现 CMDB 的基石。

11.5 自主搭建 CMDB

本节将切入主题，搭建一个适用于项目的 CMDB，当然选取的 CMDB 就是上文提到的开源解决方案 openDCIM only。我们将从安装配置展开，然后结合项目需求进行私人订制，并赋予流程化管理，以达到与 CMDB 天人合一的境地。

11.5.1 openDCIM 安装

openDCIM 是一个典型 LAMP 应用，安装起来较为方便。下面介绍一下安装步骤。

安装 rpm 包的命令如下：

```
[root@opendcim /]# yum install mysql mysql-server httpd php php-common php-cli
php-pdo php-mysql php-mbstring php-snmp php-xml
```

然后开启服务并设置开机启动，命令如下：

```
[root@opendcim /]# /etc/init.d/httpd start
[root@opendcim /]# /etc/init.d/mysqld start
[root@opendcim /]# chkconfig httpd on
[root@opendcim /]# chkconfig mysqld on
```

设置 MySQL 并创建 openDCIM 用户和数据库，命令如下：

```
[root@opendcim /]# mysql_secure_installation
```

在这一步需要实现如下功能：

- ❑ 设置 MySQL root 密码。
- ❑ 移除 MySQL 匿名账户。
- ❑ 禁止 root 远程登录。
- ❑ 删除 test database。

```
[root@opendcim /]# mysql -u root -p
mysql> create database dcim;
mysql> grant all privileges on dcim.* to 'dcim' identified by 'dcimpassword';
```

接着，设置 Apache 并创建 htpasswd 认证，命令如下：

```
[root@opendcim /]# vim /etc/httpd/conf.d/opendcim.example.com.conf
<VirtualHost *:80>
    DocumentRoot /var/www/opendcim
    ServerName opendcim.example.com
    <Directory /var/www/opendcim>
        AllowOverride All
        AuthType Basic
        AuthName "openDCIM"
        AuthUserFile /var/www/.htpasswd
        Require valid-user
    </Directory>
</VirtualHost>
[root@opendcim /]# touch /var/www/.htpasswd
[root@opendcim /]# htpasswd /var/www/.htpasswd admin
```

下载并安装 openDCIM。地址为 <http://www.opendcim.org/downloads.html>，笔者下载的是 4.0.1 版本，安装命令如下：

```
[root@opendcim /]# wget http://www.opendcim.org/packages/openDCIM-4.0.1.tar.gz -P
/var/www/
[root@opendcim /]# tar zxvf openDCIM-4.0.1.tar.gz -C /var/www/
[root@opendcim /]# ln -s /var/www/opendCIM-4.0.1 /var/www/opendcim
```

最后，配置 db 信息并重启 Apache，命令如下：

```
[root@opendcim /]# cd /var/www/opendcim
[root@opendcim opendcim]# cp db.inc.php-dist db.inc.php
[root@opendcim opendcim]# vim db.inc.php
$dbhost = 'localhost';
$dbname = 'dcim';
$dbuser = 'dcim';
$dbpass = 'dcimpassword';
[root@opendcim opendcim]# service httpd restart
```

至此，命令行的安装步骤已经完成，接下来的初始化工作，都会在网页界面上进行。

第一个界面，创建 Department，如图 11-1 所示。

第二个界面，创建 Datacenter，如图 11-2 所示。

第三个界面，创建 Cabinet（机柜），如图 11-3 所示。

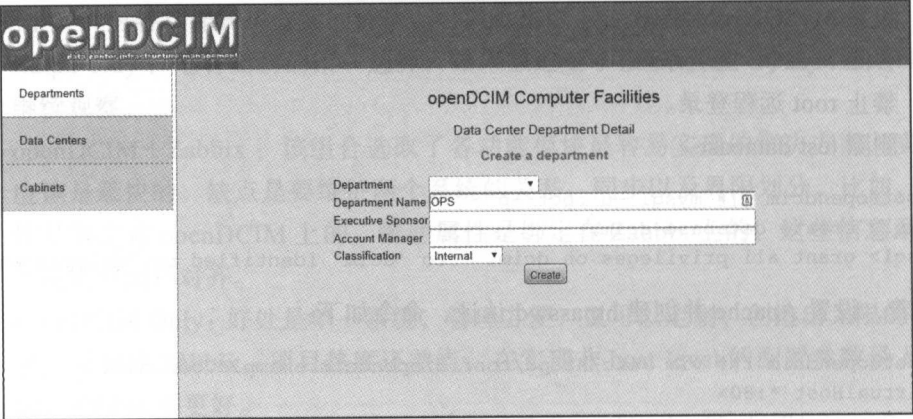


图 11-1 创建 Department

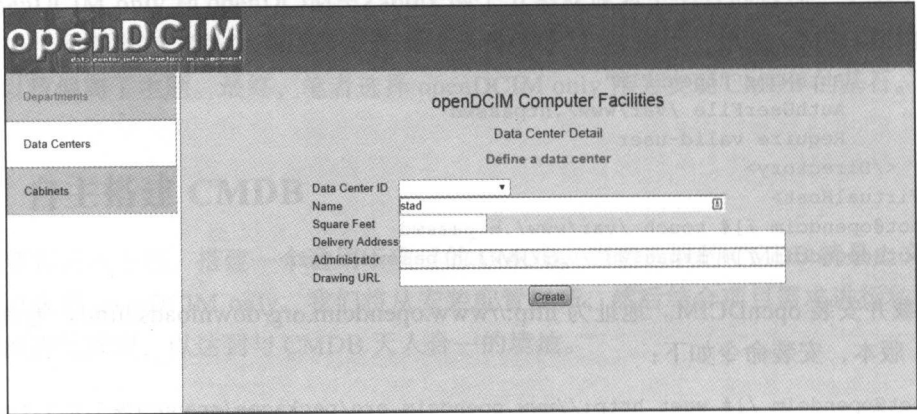


图 11-2 创建 Datacenter

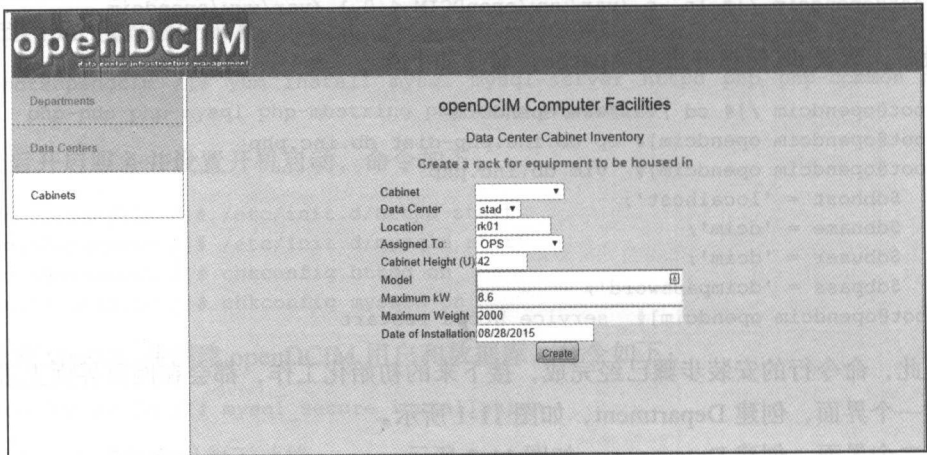


图 11-3 创建 Cabinet

第四个界面，完成创建，如图 11-4 所示。

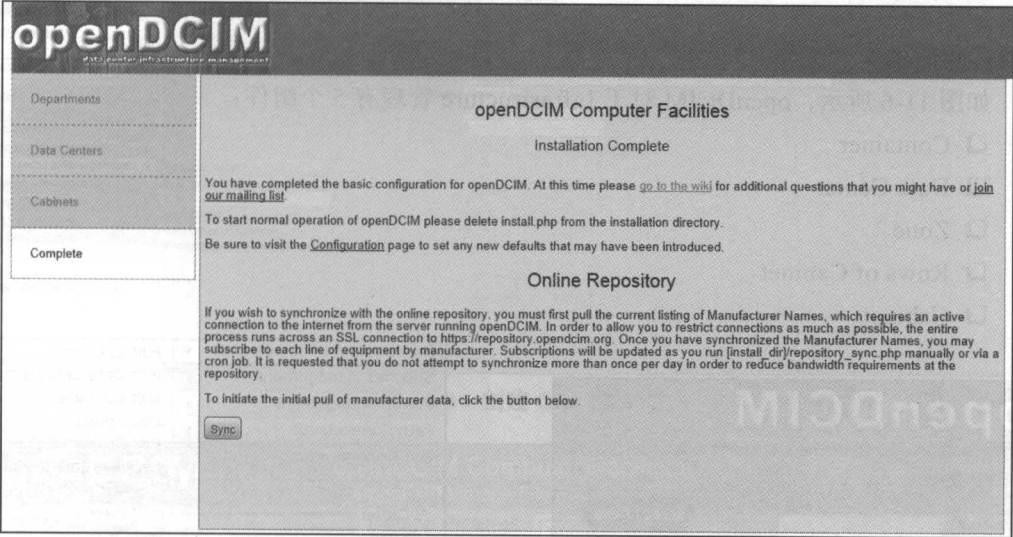


图 11-4 完成创建

第五步，重命名 install.php 为 install-bak.php，完成初始化工作，命令如下：

```
[root@opendcim-server opendcim]# mv install.php install-bak.php
```

刷新首页，如图 11-5 所示。

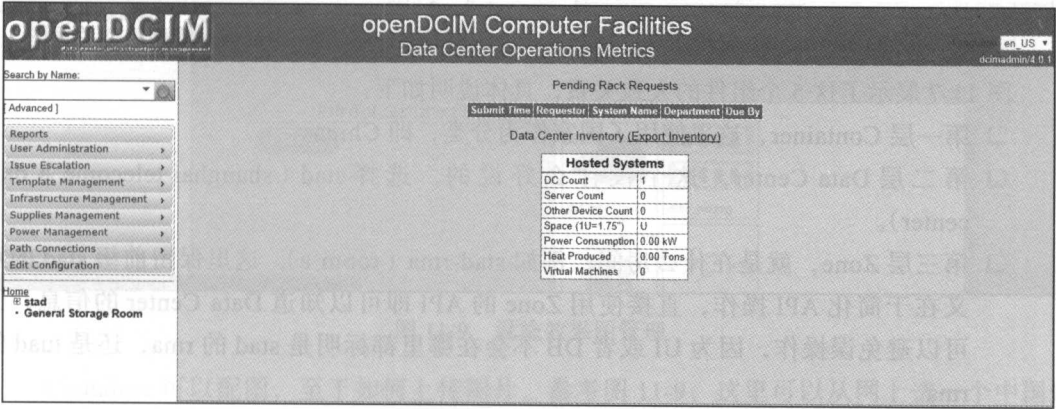


图 11-5 初始化首页

至此，openDCIM 的初始化工作已完成，接下来立即进入配置阶段。

11.5.2 openDCIM 配置

1. 了解 openDCIM 的 infrastructure 各组件关系

在上文笔者创建了第一个 Data Center（数据中心）和第一个 Cabinet（机柜），但现实生

活中，一旦管理的服务器多起来，往往需要更加细的颗粒度来描述一些 infrastructure（基础设施），比如哪个机房、第几排，而 openDCIM 恰好能满足这个需求，它提供了相应的组件来描述它们。

如图 11-6 所示，openDCIM 对于 Infrastructure 管理有 5 个组件：

- ❑ Container
- ❑ Data Center
- ❑ Zone
- ❑ Rows of Cabinet
- ❑ Cabinet

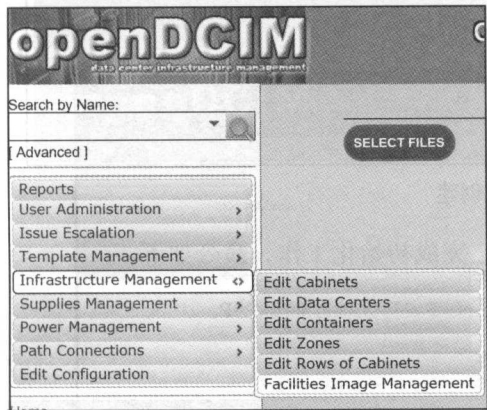


图 11-6 Infrastructure 管理组件

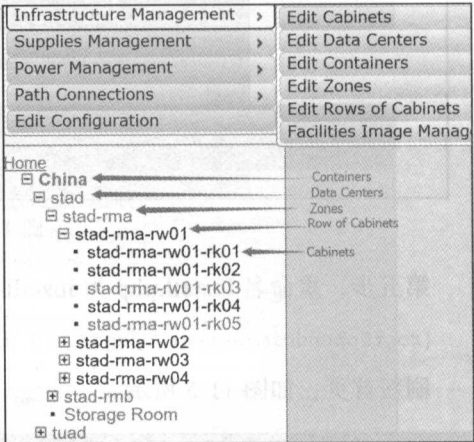


图 11-7 Infrastructure 管理组件关系图

图 11-7 展示了这 5 个组件的大致关系，具体说明如下：

- ❑ 第一层 Container，这里使用了国家作为分类，即 China。
- ❑ 第二层 Data Center，这个没什么好说的，选择 stad（shanghai telecom A data center）。
- ❑ 第三层 Zone，就是在什么房间，例如 stad-rma（room a），这里保留前缀 stad 的意义在于简化 API 操作，直接使用 Zone 的 API 即可以知道 Data Center 的信息，又可以避免误操作，因为 UI 或者 DB 不会在哪里都标明是 stad 的 rma，还是 tuad 的 rma。
- ❑ 第四层 Row of Cabinet，即第几排机柜，这里选择 stad-rma-rw01（row 01），保留前缀的意义同上。
- ❑ 第五层 Cabinet，即具体哪个机柜，这里选择 stad-rma-rw01-rk01（上海电信 A 机房，房间 a，第 01 排机柜的第 01 个 rack），保留前缀的意义同上。

2. 创建 openDCIM 的 infrastructure 各组件

（1）创建 Container，如图 11-8 所示。

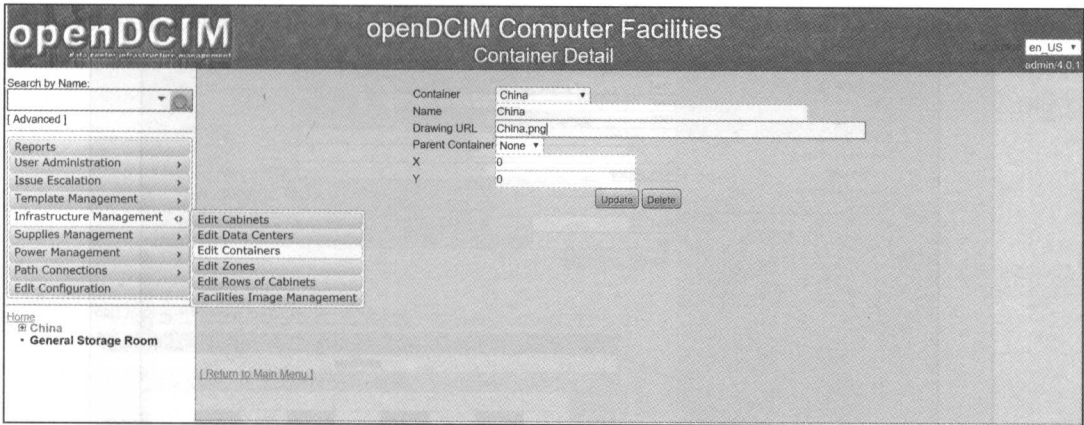


图 11-8 创建 Container

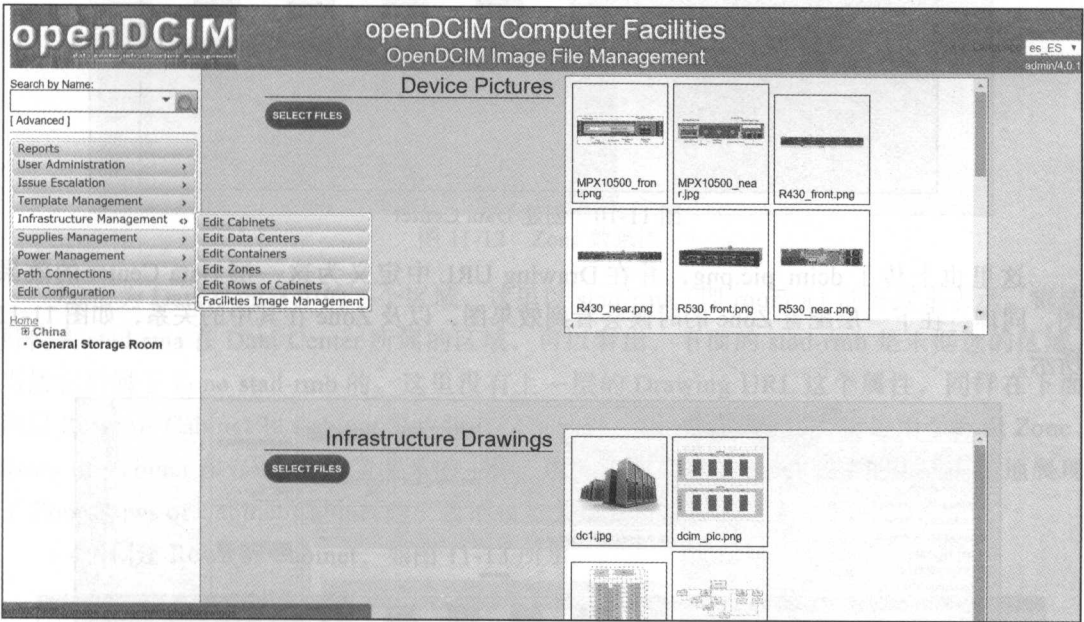


图 11-9 设施效果图管理

Container 可以配图，至于如何上传图片，参考图 11-9，这里可以从网上选一个中国地图为样图，并在 Draw URL 里选择相应的图片名称即可。

此外，可以看到图 11-8 里有一个选项叫 Parent Container，这个其实给更复杂的项目提供了可描述性，本文不会用到，置为 None。

(2) 创建 Data Center，如图 11-10 所示。

可以看到，由于上一层 Container 使用了一张中国地图，因此在创建 stad (Shanghai Telecom A Datacenter 上海电信 A 机房) 的时候，可以拖动图标，直接标识这一层 Data

Center 在 Container China 效果图中的位置。

The screenshot shows a form titled 'Data Center ID' with the following fields and values:

Field	Value
Name	stad
Square Feet	1500
Delivery Address	
Administrator	
Drawing URL	dcim_pic.png
Design Maximum (kW)	0
Container	China
X	1084
Y	580

Annotations on the form:

- An arrow points from the 'Container' field to the text '上一层的名称' (Name of the previous layer).
- An arrow points from the 'Drawing URL' field to the text 'DC 对应的图，讲解下一层 Zones 会看到效果图' (Diagram corresponding to the DC, explaining the next layer's Zones will see the effect diagram).
- At the bottom left, there is a text prompt: 'Click on the image to select DC coordinates'.

图 11-10 创建 Data Center

这里也上传了 dcim_pic.png，并在 Drawing URL 中定义为这一层 Data Center 的效果图，同理，在下一层配置 Zone 的时候会看到效果图，以及 Zone 在其中的关系，如图 11-11 所示。

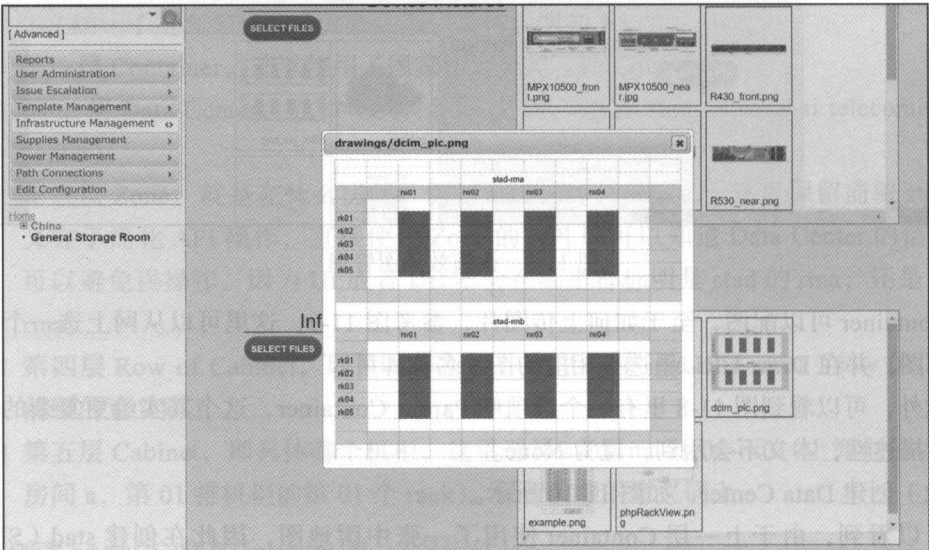


图 11-11 Data Center 效果图

(3) 创建 Zone, 如图 11-12 所示。

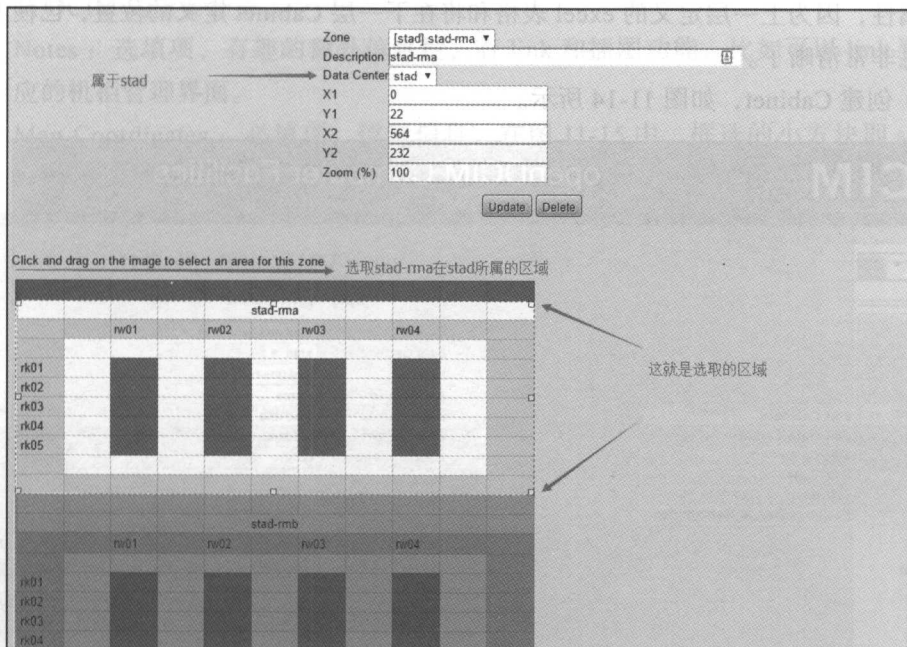


图 11-12 Zone 效果图

Zone 为 Data Center 中的一个区域, 这里以 stad-rma (即 room a) 的方式来表示。框选可代表 stad-rma 在 Data Center 所属的区域, 可以看出, 下面的 stad-rmb 是未框选的区域, 当然它是属于 Zone stad-rmb 的。这里没有上一层的 Drawing URL 这个属性, 同样在下面两层 Rows of Cabinet 和 Cabinet 里也没有这个属性, 因为这张图其实就是为了标识 Zone、Rows of Cabinet 和 Cabinet 位置关系的主体。因此该图其实以 excel 表格的方式简洁地展现了 Zone/Rows of Cabinet/Cabinet 这三者的位置关系。

(4) 创建 Rows of Cabinet, 如图 11-13 所示。

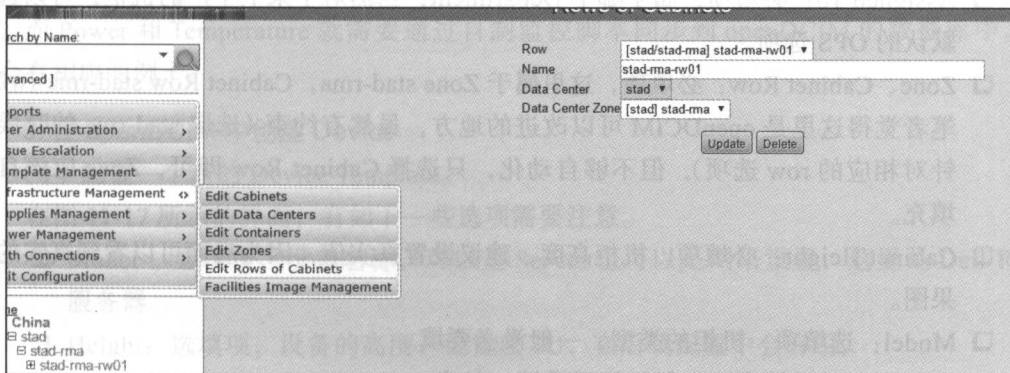


图 11-13 创建 Rows of Cabinet

这一层相当简洁，rw01 就代表了 Row 01。细心的读者可以发现，这一层没有描述位置关系的属性，因为上一层定义的 excel 表格和将在下一层 Cabinet 定义的位置，已使 Row 的位置信息非常清晰了。

(5) 创建 Cabinet，如图 11-14 所示。

图 11-14 创建 Cabinet

这一层有一些有趣的功能，值得细说，如下：

- ❑ Location：必填项，其实就是机柜名称（stad-rma-rw01-rk01），笔者也很好奇为什么是这个奇怪的属性名。
- ❑ Assigned To：必填项，属于哪个 Department，当然对于未上生产的机柜，可以选择默认的 OPS 选项。
- ❑ Zone、Cabinet Row：必填项，这里属于 Zone stad-rma，Cabinet Row stad-rma-rw01，笔者觉得这里是 openDCIM 可以改进的地方，虽然有约束（选取 stad-rma 的时候只针对相应的 row 选项），但不够自动化，只选择 Cabinet Row 即可，Zone 应该自动填充。
- ❑ Cabinet Height：必填项，机柜高度，建议设置真实值，因为后面可以看到真实的效果图。
- ❑ Model：选填项，机柜的类型，一般没必要填。
- ❑ Key/Lock：选填项，一般机柜不带锁，忽略。

- ☐ Maximum kW、Maximum Weight：选填项，建议询问机房人员获得，毕竟电力管理只 DCIM 的重要任务。
- ☐ Notes：选填项，有趣的额外信息栏，有 link 和插图功能，比如可以 link 到 DC 相应的机柜管理界面。
- ☐ Map Coordinates：必填项，位置信息，在图 11-15 中，框选的小方块即 stad-rma-rw01-rk01 所处的位置。

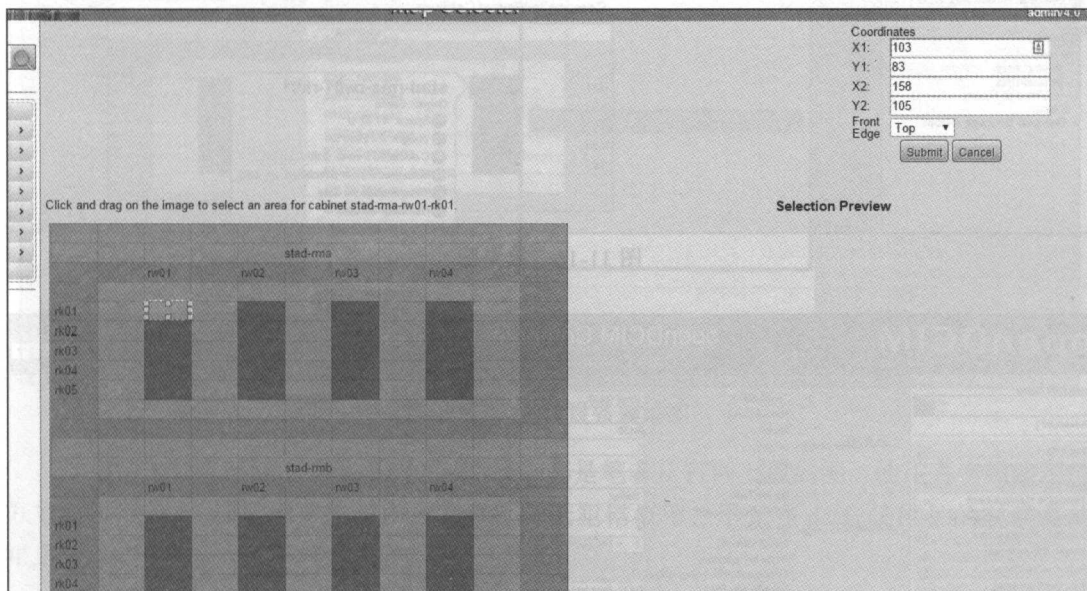


图 11-15 Cabinets 位置信息

以上是五个组件 Container、Data Center、Zone、Rows of Cabinet、Cabinet 的相应设置，在配置 server 之前，下面先来看看酷炫的效果图吧（如图 11-16 所示）。

可以看出随着鼠标的移动，机柜的基本信息都显示出来了。当然目前的信息都是静态的，Space 和 Weight 之类的问题不大，只要上架的 server 信息正确，这里会自动计算使用率，而 Power 和 Temperature 就需要通过自制监控脚本同步到 openDCIM 的数据库中，后文会有相应的例子。

3. 在 openDCIM 中创建 Device

（1）配置一个新的 Device Template。

在图 11-17 所示的界面，有如下一些选项需要注意。

- ☐ Model：必填项，设备名称，可以是 Server 也可以是网络设置，这里是 Dell R430 服务器。
- ☐ Height：选填项，设备的高度，设置成 1U，在后续配置中会有用。
- ☐ Weight：选填项，设备的重量，本章不涉及，感兴趣的读者可配置。

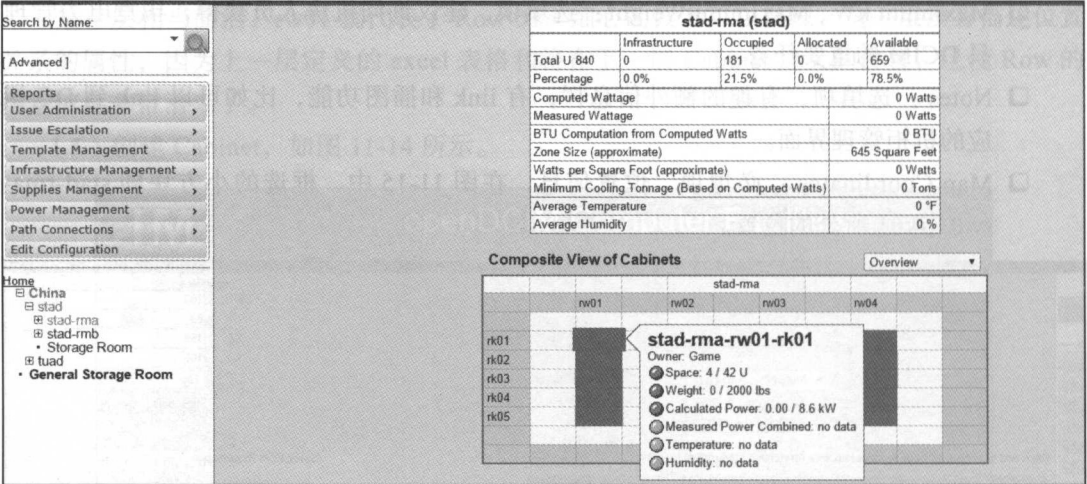


图 11-16 效果图

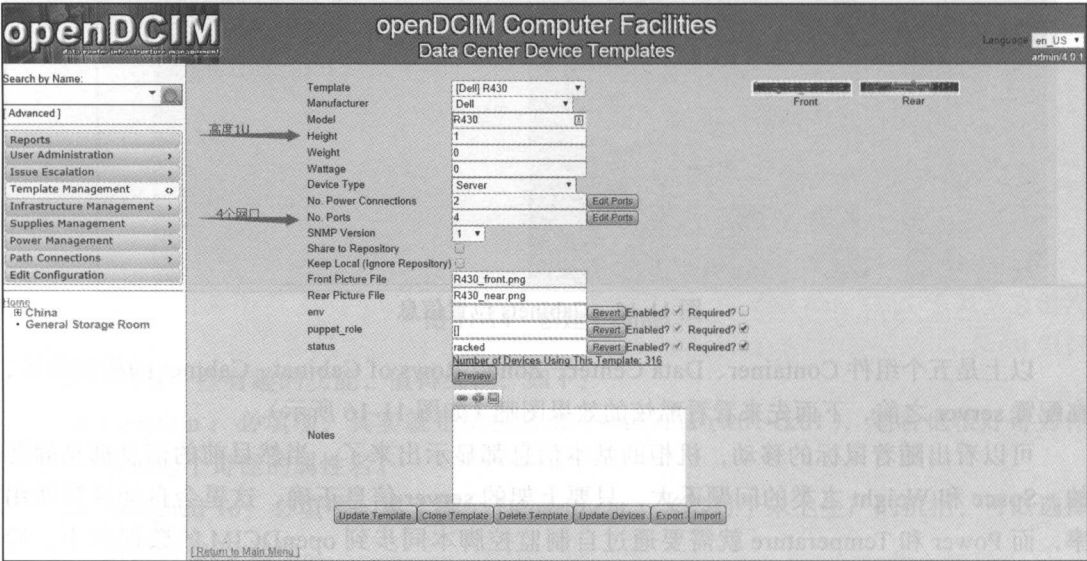


图 11-17 配置 Device Template

- ❑ Wattage: 选填项，设备的电功率即瓦数，本章不涉及，感兴趣的读者可配置。
- ❑ No. Power Connections : 选填项，设备的电源接口数量，设置成 2，在后续配置中会有用。
- ❑ No. Ports: 选填项，设备的网口数量，设置成 4，在后续配置中会有用。
- ❑ Front Picture File : 选填项，设备的前方效果图，到官网上找到该设备截图即可，并如图 11-18 所示的方式上传效果图。
- ❑ Rear Picture-File : 选填项，设备的后方效果图，到官网上找到该设备截图即可，并

如图 11-18 所示的方式上传效果图。

□ env、puppet_role、status：为自定义属性，后续会有讲解。

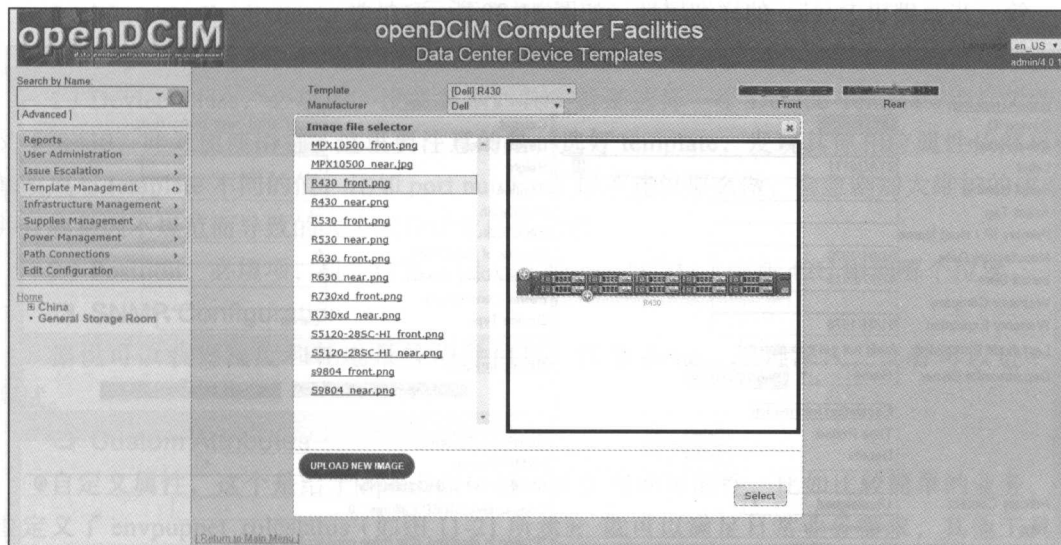


图 11-18 上传设备效果图

这里的 R430_front.png 和 R430_near.png 都是笔者从官网上截的图。使用真实缩略图的好处是可以在 dc 现场维护的时候更直观，降低犯错概率，毕竟在 dc 现场维护的时候真的很累。

(2) 配置一个新的 Device，如图 11-19 所示。

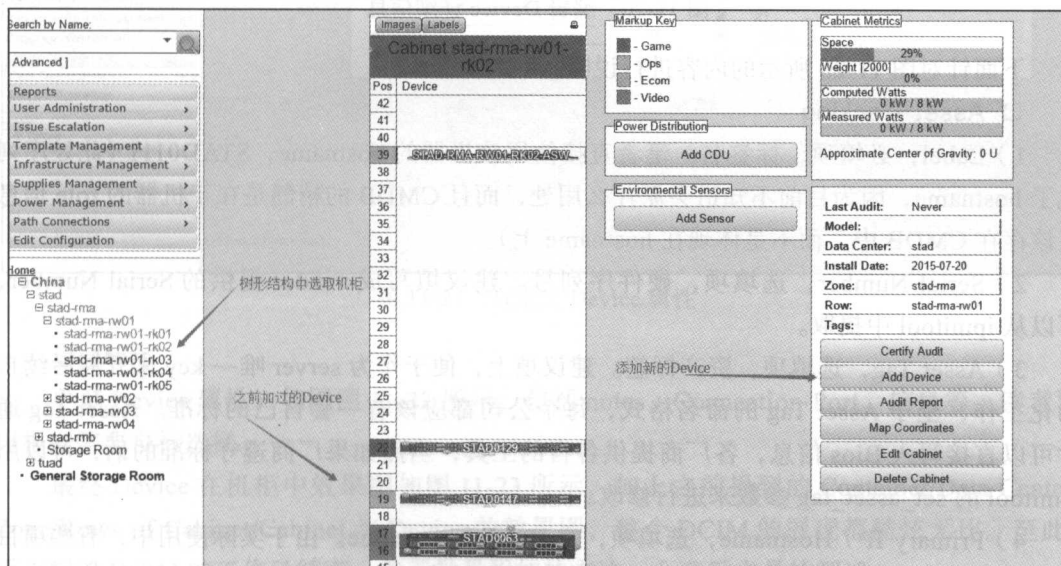


图 11-19 添加 Device

- 第一步，从树形结构中选取一个机柜，这里选取 stad-rma-rw01-rk02。
- 第二步，点击 Add Device。
- 第三步，编辑 Device 的详细属性，如图 11-20 所示。

Asset Tracking

Device ID

587

Reservation?

☐

Label

STAD0111

Serial Number

Asset Tag

Primary IP / Host Name

Manufacture Date

01/01/1970

Install Date

07/27/2015

Warranty Company

Warranty Expiration

01/01/1970

Last Audit Completed

Audit not yet completed

Departmental Owner

Game

Show Contacts

Escalation Information

Time Period

Select...

Details

Select...

Primary Contact

Unassigned

Tags

Awaiting input...

Custom Attributes

env

prod

puppet_role

status

live

Preview

Physical Infrastructure

Cabinet

stad / stad-rma-rw01-rk02

Device Class

Dell - R430

Height

1

Position

13

Half Depth

☐

Back Side

☐

Number of Data Ports

4

Nominal Draw (Watts)

0

Power Connections

2

Device Type

Server

Device Images

SNMP Configuration

SNMP Version

2c

SNMP Read Only Community

Consecutive SNMP Failures*

0

*Polling is disabled after three consecutive failures.

VMWare ESX Server Information

ESX Server?

False

图 11-20 编辑 Device 详细信息

下面针对图 11-20 所示的内容进行说明。

❑ Asset Tracking

- 1) Label，必填项，标签名。笔者直接使用该机器的 hostname，STAD0111（此处序列化了 hostname，因为目前不知道会派什么用处，而且 CMDB 的精髓是在于机器的 role 信息应该存在 CMDB 中，而不是体现在 hostname 上）。
- 2) Serial Number，选填项，硬件序列号。建议填写成 vendor 提供的 Serial Number，可以从 ipmitool 中提取。
- 3) Asset Tag，选填项，资产标签。建议填上，便于作为 server 唯一 key 来开展后续自动化工作，至于 Asset Tag 的命名格式，每个公司都应该有一套自己的标准。Asset Tag 通常可以直接写入 Bios 信息，各厂商提供各自的工具，当然如果厂商遵守标准的话，可以用 ipmitool 的 set_asset_tag 参数来进行修改。
- 4) Primary IP / Hostname，选填项，主 IP 或者 hostname。由于实际使用中，有些项目是先让 DHCP 分配 IP，再绑定 Mac 的方式来固定 Device IP 的，因此此处也可以使用相应

的步骤从 DHCP 里获得 MAC，并且通过 ipmitool 来获得相应机器的 MAC，从而查询获得 Device IP，再回写到 openDCIM 的方式来填写。

5) Warranty Expiration，选填项，质保期限。建议填上，便于后续审计提醒。

❑ Physical Infrastructure

1) Device Class，必填项，设备类型，其实就是选择一个 Device Template，上文已定义了 R430，此处便使用它。这里要注意的是，选好 template，发现其他相应属性依然可以修改成与 template 不同的值，比如 port number，但不建议那么做，会急剧加大维护的成本，并且容易因不规范而导致的人为或自动化任务出错。

2) Position，必填项，机柜位置。用于告诉 openDCIM，应在 42U 中的哪个位置。

❑ SNMP Configuration

据说可以获得温度和耗电量等硬件信息，作为 demo，不再深入，读者可以进一步测试。

❑ Custom Attributes

自定义属性，这个是给予 openDCIM 进一步扩展的可能性，比如比较简单的业务，只要定义了 envpuppet_rolestatus（如图 11-21 所示），就可以满足日常业务需求，其他工具可以根据这些属性进行自动化任务。上文提到的 MAC 地址，也是一个常用的信息，可以通过 ipmitool 录入。

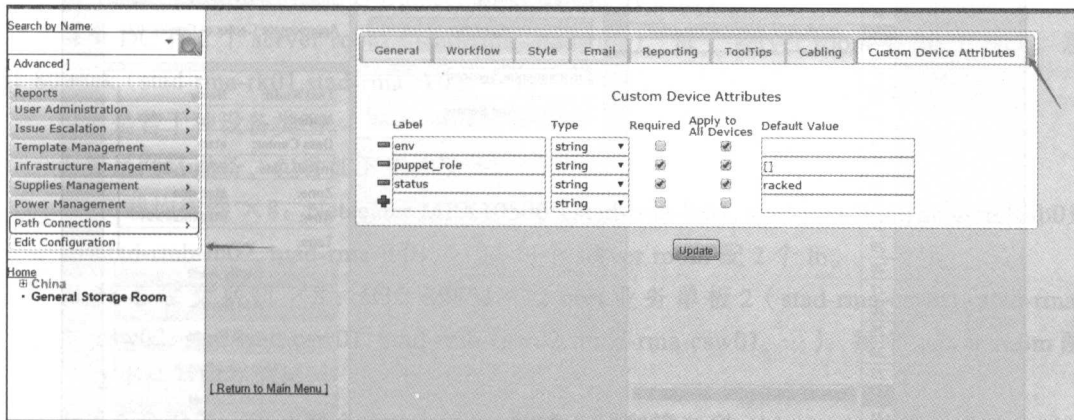


图 11-21 自定义 Device 属性

❑ More

更多 Device 属性，如图 11-22 所示，包括 notes、Connection Port、信息等，读者可以根据需要自行选择。

最终 Device 在机柜中效果图如图 11-23 所示，加上之前提到的 Container/Data Center/Zone/Rows of Cabinet/Cabinet 与 Device 的效果图，整个 DCIM 的展现都酷炫无比。至此，手工配置 DCIM 的工作已结束，关于批量添加的方法，会在后文具体阐述。

Preview

Notes

Operational Log

Date

Add note

Power Connections

#	Port Name	Device	Device Port	Notes
1	Power Connection 1			
2	Power Connection 2			

Limit device selection to:

Row

Zone

Datacenter

Global

Connections

#	Port Name	Device	Device Port	Notes	Media Type	Color Code
1	Port1					
2	Port2					
3	Port3					
4	Port4					

Update

Copy

Certify Audit

Delete

Export Connections

Print

Log View

图 11-22 更多 Device 属性

Pos

Device

42

41

40

39

38

37

36

35

34

33

32

31

30

29

28

27

26

25

24

23

22

21

20

19

18

17

16

15

14

13

12

11

10

9

rk02

STAD-RMA-RW01-RK02-ASW

STAD0129

STAD0147

STAD0063

STAD0111

STAD0093

- Ops

- Ecom

- Video

Power Distribution

Add CDU

Environmental Sensors

Add Sensor

Weight [2000]

29%

0%

Computed Watts

0 kW / 8 kW

Measured Watts

0 kW / 8 kW

Approximate Center of Gravity: 0 U

Last Audit: Never

Model:

Data Center: stad

Install Date: 2015-07-20

Zone: stad-rma

Row: stad-rma-rw01

Tags:

Certify Audit

Add Device

Audit Report

Map Coordinates

Edit Cabinet

Delete Cabinet

点击可以跳转至Device界面

图 11-23 Device 机柜效果图

11.5.3 openDCIM API

上文在 CMDB 选型的时候曾提到过, API 是 CMDB 的必要条件, 因此本节将以一个现实例子为命题, 通过介绍 openDCIM 的 API, 最后针对这个现实例子, 举例说明 openDCIM API 的常见应用场景。

1. 一个现实的例子

以下场景, 是一个综合性互联网公司的典型案例。

3 个业务如下:

- ☐ 商城 - Ecom
- ☐ 游戏 - Game
- ☐ 视频 - Video

每个业务有如下 3 种环境:

- ☐ env - Prod
- ☐ env - Test
- ☐ env - Dev

2 个 IDC 如下:

- ☐ stad (Shanghai Telecom A Datacenter 上海电信 A 机房)
- ☐ tuad (Tianjin Unicom B Datacenter 天津网通 A 机房)

每个 DC 有 2 个 server room (stad-rma, stad-rma), 每个 server room 有 20 个 rack, 总计 80 rack (stad-rma-rk01, stad-rma-rk02, ...)。

以下是若干种设备类型。

一类是网络设备, 包含如下三种设备。

- ☐ 负载均衡器 × 8: Netscaler MPX10500 (stad-rma-lb01, stad-rma-lb02, stad-rmb-lb01, stad-rmb-lb02, tuad-rma-lb01, ...), 每个 server room 配 2 个 lb。
- ☐ 核心层交换机 × 8: H3C S9804 - 12 port 业务单板 2 (stad-rma-csw01, stad-rma-csw02, stad-rmb-csw01, stad-rmb-csw02, tuad-rma-csw01, ...), 每个 server room 配 2 个核心层交换机。

上述 2 个设备, 放在每个 server room 的第一列和第二列 rack group 的第一个 rack 上, 即 ***d-rm*-rw01-rk01 和 ***d-rm*-rw02-rk01, 这些 rack 不放服务器。由于这 2 个设置为 room 为单位, 因此命名方式也以 room 为截止字段。总计 8 个 rack 是网络设备专属, 72 个 rack 上放服务器。

- ☐ 接入层交换机 × 72: H3C S5120-28SC-HI - 24 port (stad-rma-rw03-rk01-asw, stad-rma-rw04-rk01-asw, ...)。

由于核心交换机在 ***d-rm*-rw01-rk01 和 ***d-rm*-rw02-rk01 上, 因此这些 rack 的接入层交换机就省掉了。所以接入层交换机和 rack 是一一对应的关系, 总计 72 台, 命名方式

也以 rack 为截止字段。

二是服务器。服务器的位置尽量做到分摊风险原则，平均分布在每个 rack、每个 rack group（机柜列，Cabinet Row）以及每个 room 里。

- ❑ 存储型服务器 ×24：Dell R730XD（stad0001~stad0012, tuad0001~tuad0012），每个 room 6 个，每个 rack group 1~2 个，每个 rack 0~1 个。
- ❑ DB 型服务器 ×64：Dell R630（stad0013~stad0044, tuad0013~tuad0044），每个 room 16 个，每个 rack group 4 个，每个 rack 0~1 个。
- ❑ 虚拟化型服务器 ×96：Dell R530（stad0045~stad0092, tuad0045~tuad0092），每个 room 24 个，每个 rack group 6 个，每个 rack 1~2 个。
- ❑ app 型服务器 ×316：Dell R430（stad0093~stad0250, tuad0093~tuad0250），每个 room 79 个，每个 rack 组列 19~20 个，每个 rack 4~5 个。



服务器不使用类似 stad-rma-rw01-rk01-srv01 命名规范的原因是，系统管理员操作服务器频繁，但不需要把机房的物理信息记得如此清楚，实在需要的情况可以到 CMDB 中提取，而使用物理层级命名规范的主要原因也是为了网络管理员管理设备清晰可见。

下面是三种 config 录入 level。

❑ dc level

- Zabbix URL：监控地址。
- DNS Server IP：DNS 服务器 IP。

❑ Project+ENV Level

- App Version：APP 版本信息。
- 日志级别：比如各种 project 的 dev 环境都是 debug，但 prod 环境以有些更新频繁的项目是 info，有些稳定的 project 是 warn。

❑ host level

- Rack：表示在哪个机柜上。
- 设备类型：比如是 DELL R430 还是 Netscaler MPX10500。
- 每个 interface 的 IP：gateway、network 等基础属性。
- Project：所属的业务。
- Env：所属的环境，比如是 dev 还是 prod。
- Role：服务器角色，比如可以在执行 Puppet 的时候映射到 Puppet 的 module。



这里提到了 Project+ENV Level 的这种组合，虽然用户可以 Project/ENV/ENV+Role/ENV+DC/Project+DC/Project+Role/Role+DC/Project+ENV+Role 这些组合的 level，但是在笔者的实践过程中，发觉在使用 CMDB 的过程中必须做到一定的

平衡，不能一味地按照写程序的思路（任何出现 2 次的 code 要写成 function/class）来进行，要考虑易用性和便于理解与交流。比如 game_dev 的 db 的 mysql 版本以及其 os 版本，没必要创造出 Project+Env+Role Level 专门录入这些配置，直接在 Project+Env Level 上用变量前缀为 db 方式即可，比如 db_mysql_version、db_os_version 等。

上述现实的例子，乍一看，不少人都会头大，但是经过提炼后，就会得到需求列表，见表 11-4。

表 11-4 现实示例的需求分析

现实示例	openDCIM 中的实现方法	API
3 个 Project	Device 默认属性 Departmental Owner	否
3 个 Env	Device 自定义属性 Custom Attributes 中的 env	否
2 个 Dc	使用 openDCIM 中的默认 Data Center 对象	否
4 个 Server Room	使用 openDCIM 中的默认 Zone 对象	否
80 个 Rack	使用 openDCIM 中的默认 Cabinet 对象	是，数量较多
7 种 Device Type	使用 openDCIM 中的默认 Device Template 功能	否
500 台 Server+88 台网络设备	使用 openDCIM 中的默认 Device 对象	是，数量较多
2 个 Dc_Level+2 个 Project+Env Level 的 Conf	使用 openDCIM 中的默认 Device 对象，但命名以前缀 conf_LevelName（conf_stad，conf_game_dev）来规范，并且以相应的 Departmental Owner、env 和 Datacenter 来关联相应的 host。之后 conf 管理，使用 Device 自定义属性 Custom Attributes 的功能	是，虽然数量不多，但这是 CMDB 的核心，让其他工具通过 API 查询各 level 的 CI（Configuration Item）
若干 Host Level 的 Conf	Device 自定义属性 Custom Attributes 中的 env	是，数量较多，且需要让其他工具通过 API 查询，比如 Puppet

从表 11-4 可以看出，学习 API 后，对于批量操作的需求，可以从容应对。在工具中写入如何关联非 host level conf 的逻辑后，openDCIM 可以称得上一个称职的 CMDB 工具。下文中，笔者将会从如何调用 API 入手，结合现实例子，来介绍 openDCIM API 的使用。

2. API 认证：Authentication

在上文 openDCIM 的安装过程中，有一个 htaccess 的安装步骤，用来设置网站登录密码，事实上，默认的 API 登录密码也是它。示例命令如下：

```
[root@opendcim /]# httpasswd /var/www/.htpasswd admin
```

但从 4.2 版本开始，openDCIM 加入了 ldap 认证和 user_key 的认证方式，只要在 UI 界面 openDCIM User Manager 产生 API Key，并且在访问的 header 里加入 UserID 和 API Key 即可，由于笔者手头实验环境没有 ldap，因此没有实现，有兴趣的读者可以看官方文档

实现。

本文就直接使用 `htpasswd` 以简化 API 调用，2 种示例方法如下：

```
[root@opendcim /]# curl -s -u admin:adminpassword 127.0.0.1/api/v1/department
[root@opendcim /]# curl -s -H 'UserID:xxxx' -H 'APIKey:xxxx' 127.0.0.1/api/v1/department
```

3. API 查询：GET

认证过后，第一个要关注的 API 就是 GET，毕竟熟悉了 GET 后，才知道有哪些属性，规范是怎麼样的，才可以从容地使用修改的 API。

GET ALL 的代码如下：

```
[root@opendcim /]# curl -s -u admin:admin 127.0.0.1/api/v1/department | python
-m json.tool
{
  "department": [
    {
      "Classification": "Online",
      "DeptColor": "#C9C430",
      "DeptID": "3",
      "ExecSponsor": "",
      "Name": "Ecom",
      "SDM": ""
    },
    {
      "Classification": "Online",
      "DeptColor": "#007FF5",
      "DeptID": "2",
      "ExecSponsor": "",
      "Name": "Game",
      "SDM": ""
    },
    {
      "Classification": "Internal",
      "DeptColor": "#54DB33",
      "DeptID": "1",
      "ExecSponsor": "",
      "Name": "Ops",
      "SDM": ""
    },
    {
      "Classification": "Online",
      "DeptColor": "#50ACBA",
      "DeptID": "4",
      "ExecSponsor": "",
      "Name": "Video",
      "SDM": ""
    }
  ],
  "error": false,
```

```

    "errorcode": 200
}

```

上述代码的说明如下：

- ❑ `api/v1/department`, `api/v1/` 固定的 api URL, 而 `department` 是要查询的对象类型。
- ❑ `python -m json.tool` 是一个利用 Python json 模块, 用 bash 管道, 输出漂亮 json 格式的工具, 在 bash 中, 所有默认 json 输出都是挤在一块的, 难以阅读。
- ❑ Output 是一个 hash, 有 3 个 key。error 表示是否有错误, errorcode 的返回值 200 代表运行是正常的; department 表示请求的 output 的内容, 是一个 array, 每个 element 是一个 hash; Name 为部门名字, 在这个例子中, 把这个属性影射为 Project, 如 Video/Game/Ecom/Ops; DeptID 为部门 ID, 也就是 project ID, 在 device 的 API output 中会使用到。

除了 department, 4.0.1 版本还支持如下 API 入口。

- ❑ `/datacenter`: 机房。
- ❑ `/zone`: 区域, 也就是 server room。
- ❑ `/cabrow`: 机柜行, 也就是 rack group。
- ❑ `/cabinet`: 机柜。
- ❑ `/device`: 设备, 服务器或者网络设备。
- ❑ `/devicetemplate`: 设备 template, 可以是 DELL R430 或者 Netscaler MPX10500。
- ❑ `/manufacturer`: 供应商。

使用 GET 方法时, openDCIM 还提供一些预设的 search, 代码如下:

```

[root@opendcim /]# curl -s -u admin:adminpassword 127.0.0.1/api/v1/device/1 |
python -m json.tool
[root@opendcim /]# curl -s -u admin:adminpassword 127.0.0.1/api/v1/device/
bydatacenter/2 | python -m json.tool

```

上述代码的说明如下:

- ❑ 第一个格式, 就是加上了 deviceid (/1) 作为 search 条件。
- ❑ 第二个格式, 就是加上了 datacenter 的 datacenterid (/bydatacenter/2) 作为 search 条件。

这里的 2 个 search 条件都是 id, 如果要知道 id 和 name 的对应列表要再查一次 datacenter 或者 device, 所以建议用 Python 进行 json load, 并且分析是最好的选择。

以下是 4.0.1 版本还支持预设 search。

- ❑ `/cabinet/bydc/:datacenterid`
- ❑ `/cabinet/bydept/:deptid`
- ❑ `/cabinet/:cabinetid`
- ❑ `/cabinet/:cabinetid/sensor`

- ❑ /datacenter/:id
- ❑ /device/bydatacenter/:datacenterid
- ❑ /device/:deviceid
- ❑ /device/:deviceid/getpicture
- ❑ /device/:deviceid/getsensorreadings
- ❑ /deviceport/:deviceid
- ❑ /deviceport/:deviceid/patchcandidates
- ❑ /devicetemplate/:templateid
- ❑ /devicetemplate/:templateid/dataport
- ❑ /devicetemplate/:templateid/dataport/:portnumber
- ❑ /devicetemplate/:templateid/powerport
- ❑ /devicetemplate/:templateid/slot
- ❑ /powerport/:deviceid
- ❑ /zone/:zoneid

从上述预设的 search 可以看的出来, 设计是比较粗糙的, 一些很通常的 search 条件都没有, 比如想看所有属于 game 的服务器, 这时, 默认的 search 就无力了, 必须要将所有的 device 读取到 Python 里, 并进行分析才可以实现。那么有没有更好的办法呢? openDCIM 的 4.2 版本之后给了我们答案。

使用 GET 方式时, openDCIM 还提供了自定义 search, 代码如下:

```
[root@opendcim /]# curl -s -u admin:adminpassword "127.0.0.1/api/v1/device?LAB
EL=STAD0250&Cabinet=40" | python -m json.tool
{
  "device": [
    {
      "AssetTag": "",
      "Cabinet": 40,
      "ChassisSlots": 0,
      "CustomValues": {
        "1": "[]",
        "2": "maintenance",
        "3": ""
      },
      "DeviceID": 225,
      "DeviceType": "Server",
      "ESX": 0,
      "EscalationID": 0,
      "EscalationTimeID": 0,
      "FirstPortNum": 0,
      "HalfDepth": 0,
      "Height": 1,
      "InstallDate": "2015-07-26",
      "Label": "STAD0250",
```

```

...
...
...
    }
    ],
    "error": false,
    "errorcode": 200
  }

```

以下是代码说明：

- ❑ ?LABEL=STAD0250：问号是 URL 参数的开始符，LABEL=STAD0250 是输入的参数。
- ❑ &Cabinet=40：& 是参数之间的间隔符，本例是为了介绍才给出此条件，其实，要搜索 STAD0250 这台服务器，前一个条件已经足矣。
- ❑ 其他属性同样可以作为搜索条件，查一台后仔细阅读 output，就可以举一反三。

同样，很多属性都是以 id 方式存在，需要多查询一次相应属性，并依靠 python 分析出可读的 name。

目前稳定版的 openDCIM 还缺少如下 2 个特性：

- ❑ wildcard：通配符搜索，类似 MySQL 的 Like。
- ❑ filter：过滤器，只返回相应的字段，而不是现在 select *。

笔者在写这章的时候，openDCIM 正在开发 4.3 版本，有兴趣的读者可以查看他们的 GitHub。应该会在 2016 年夏天推出，调用方法如下：

```
/api/v1/device?wildcards&Label=%%test%%
```

该方法的说明如下：

- ❑ wildcard：表示这个 URL 是为了通配符搜索。
- ❑ %%test%%：2 个 % 是分界符，当中的 test 是 search 字段。

```
/api/v1/device?wildcards&Label=%%test%%&attributes=DeviceID,Label
```

该方法的说明如下：

- ❑ attributes：表示这个 URL 期待有 filter，过滤想要的字段。
- ❑ DeviceID,Label：以英文逗号分隔，为字段的 array。

4. API 的创建：PUT

在现实工作的需求分析中，有 2 个类型需要 API 导入：第一个是 device，第二个是 rack。很可惜 openDCIM 目前版本还不支持 rack 的 POST API，好在它的数据库结构比较简单。关于 rack，目前只能以 SQL 的方式批量导入，这里着重介绍 device POST 的 API。

(1) Rack 批量导入

Rack 在 openDCIM 中的数据结构如下：


```
mysql> select * from fac_Cabinet limit 1 \G
*****1. row *****
  CabinetID: 1
  DataCenterID: 1
  Location: stad-rma-rw01-rk01
  LocationSortable: stad-rma-rw01-rk01
  AssignedTo: 1
  ZoneID: 1
  CabRowID: 1
  CabinetHeight: 42
  Model:
  Keylock:
  MaxKW: 8.6
  MaxWeight: 2000
  InstallationDate: 2015-07-20
  MapX1: 103
  MapX2: 158
  FrontEdge: Top
  MapY1: 83
  MapY2: 105
  Notes:
  UIPosition: Default
```

对上述代码的说明如下：

- ❑ 大部分属性在上文 UI 创建 rack 的时候已经阐述过，可以参考上文说明。
- ❑ 可以看到所有属性都是 ID，脚本需要有 mapping 的工作。
- ❑ MapX1、MapX2、MapY1、MapY2 代表了这个 rack 在图中的位置，如图 11-24 所示，其实代表了这个长方形在整个图中像素的位置，以下示例设为默认值 0，因为如果要完全脚本化的话，要考虑 rack 与 rack 之间的间隙转换成像素，还要考虑位置的左右上下对齐，编写的代码为了满足这些需求还是有一定难度的，建议 UI 花时间搞定。

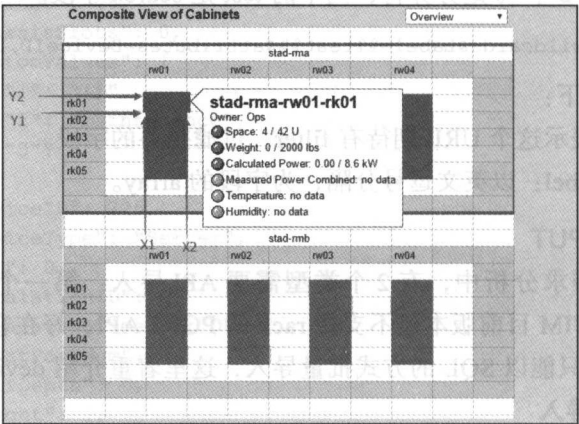


图 11-24 MapX1、MapX2、MapY1 的含义

来看看 rack 导入的需求定义。通常，当 dc 部门需要新加 rack 之前，会先查看现有情况，如图 11-25 所示的 rack 组里现有的 rack 数量，目前有 5 个 rack，希望增加到 8 个。再说直白点，表示需要加入下述三个机柜：

- ❑ stad-rma-rw01-rk06
- ❑ stad-rma-rw01-rk07
- ❑ stad-rma-rw01-rk08

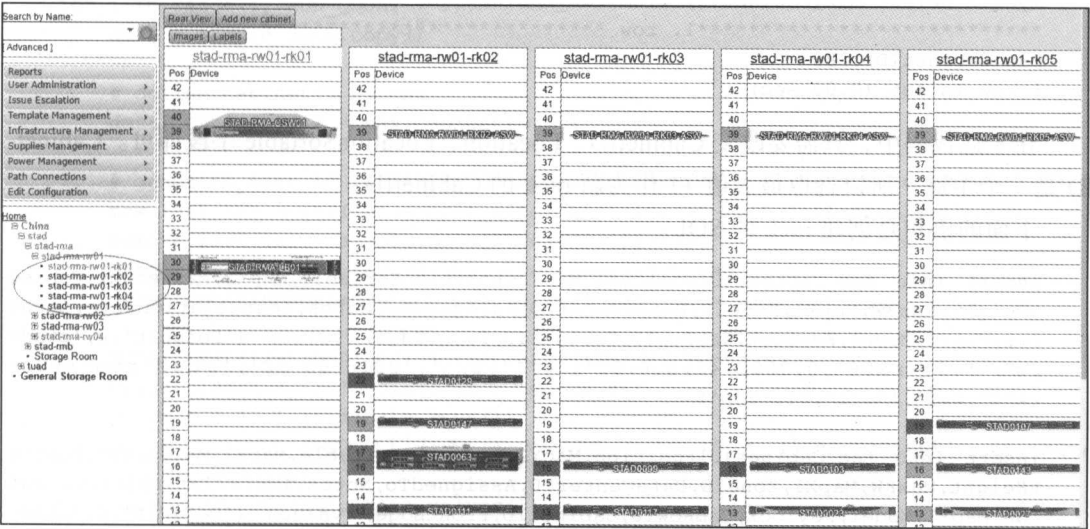


图 11-25 rack 组里现有 rack 数量

下面是 rack 的表结构分析。先来看看 stad-rma-rw01-rk05 的表结构，如下：

```
mysql> select * from fac_Cabinet where Location='stad-rma-rw01-rk05' \G
*****1. row *****
CabinetID: 5
DataCenterID: 1
Location: stad-rma-rw01-rk05
LocationSortable: stad-rma-rw01-rk05
AssignedTo: 2
ZoneID: 1
CabRowID: 1
CabinetHeight: 42
Model:
Keylock:
MaxKW: 8.6
MaxWeight: 2000
InstallationDate: 0000-00-00
MapX1: 103
MapX2: 157
FrontEdge: Top
MapY1: 167
```

```

MapY2: 190
Notes:
U1Position: Default
1 row in set (0.00 sec)

```

最大 CabinetID 如下:

```

mysql> select max(CabinetID) from fac_Cabinet \G
*****1. row *****
max(CabinetID): 80
1 row in set (0.00 sec)

```

在上述代码中,只需要改变 CabinetID、Location、LocationSortable 字段即可。CabinetID 是一个主键,所以选取最大值 +1 作为新 rack 的 CabinetID。

下面的语句是创建 rack 的 SQL。

```

insert into fac_Cabinet(U1Position,MapY2,LocationSortable,MapX2,MaxWeight,Cabine
tHeight,MaxKW,MapX1,ZoneID,DataCenterID,AssignedTo,Notes,InstallationDate,Locat
ion,MapY1,Model,Keylock,CabRowID,CabinetID,FrontEdge) values ('Default',0,'stad-
rma-rw01-rk06',0,2000,42,8.6,0,'1','1',1,'','2016-05-22','stad-rma-rw01-
rk06',0,'','','1',81,'Top')

```

```

insert into fac_Cabinet(U1Position,MapY2,LocationSortable,MapX2,MaxWeight,Cabine
tHeight,MaxKW,MapX1,ZoneID,DataCenterID,AssignedTo,Notes,InstallationDate,Locat
ion,MapY1,Model,Keylock,CabRowID,CabinetID,FrontEdge) values ('Default',0,'stad-
rma-rw01-rk07',0,2000,42,8.6,0,'1','1',1,'','2016-05-22','stad-rma-rw01-
rk07',0,'','','1',82,'Top')

```

```

insert into fac_Cabinet(U1Position,MapY2,LocationSortable,MapX2,MaxWeight,Cabine
tHeight,MaxKW,MapX1,ZoneID,DataCenterID,AssignedTo,Notes,InstallationDate,Locat
ion,MapY1,Model,Keylock,CabRowID,CabinetID,FrontEdge) values ('Default',0,'stad-
rma-rw01-rk08',0,2000,42,8.6,0,'1','1',1,'','2016-05-22','stad-rma-rw01-
rk08',0,'','','1',83,'Top')

```

当然,上述例子只是创建了 3 个 rack,如果要批量导入 rack 信息的话,可以用 bash 或者 Python 进行循环,循环时 CabinetID、Location 的 ID 会进行自增迭代。

下面来看一个 rack 的简单导入脚本。

```

import MySQLdb
import time

TARGET_CABROW = "stad-rma-rw01"
ADD_COUNT = 3

conn = MySQLdb.connect(host='127.0.0.1',user='dcim',passwd='dcimpassword',db='dc
im',port=3306)
cursor = conn.cursor(MySQLdb.cursors.DictCursor)

```

```

# get max CabinetID in global
cursor.execute('select max(CabinetID) from fac_Cabinet')
max_CabinetID = cursor.fetchone()['max(CabinetID)']

# select max CabinetID in TARGET_CABROW(stad-rma-rw01)
cursor.execute('select from fac_Cabinet where Location like "' + TARGET_CABROW +
               '%" ORDER BY CabinetID DESC LIMIT 1')
sample_data = cursor.fetchone()

for i in range(1, ADD_COUNT + 1):
    data = sample_data.copy()

    # define CabinetID by max
    data['CabinetID'] = max_CabinetID + i

    # generate next id in TARGET_CABROW(stad-rma-rw01), like stad-rma-rw01-rack06
    new_last_two = str(int(sample_data['Location'][-2:]) + i).zfill(2)
    remove_last_two = sample_data['Location'][:-2]
    data['Location'] = remove_last_two + new_last_two
    data['LocationSortable'] = remove_last_two + new_last_two

    data['InstallationDate'] = time.strftime("%Y-%m-%d")
    placeholders = ', '.join(['%s'] * len(data))
    columns = ', '.join(data.keys())
    sql = "INSERT INTO %s ( %s ) VALUES ( %s )" % ('fac_Cabinet', columns,
                                                    placeholders)
    cursor.execute(sql, data.values())
    conn.commit()
    print cursor._last_executed

cursor.close()

```

这个是一个非常短平快的 Python 脚本，把 error handler、logging、参数录入等功能都去掉了。通过 select stad-rma-rw01 中最大的 CabinetID 来产生一个 sample_data。从这个 sample_data 中产生相应的 data，之后 insert 到 MySQL 中。验证结果如下：

```

mysql> select CabinetID,Location from fac_Cabinet where Location like 'stad-rma-
      rw01%';
+-----+-----+
| CabinetID | Location          |
+-----+-----+
|          1 | stad-rma-rw01-rk01 |
|          2 | stad-rma-rw01-rk02 |
|          3 | stad-rma-rw01-rk03 |
|          4 | stad-rma-rw01-rk04 |
|          5 | stad-rma-rw01-rk05 |
|         81 | stad-rma-rw01-rk06 |
|         82 | stad-rma-rw01-rk07 |
|         83 | stad-rma-rw01-rk08 |
+-----+-----+

```

8 rows in set (0.00 sec)

(2) Device 批量导入

导入代码如下：

```
[root@opendcim /]# curl -s -u admin:adminpassword -X PUT -d Cabinet=1
'127.0.0.1/api/v1/device/stad9999' | python -m json.tool
{
  "device": {
    "AssetTag": "",
    "AuditStamp": "0000-00-00 00:00:00",
    "BackSide": 0,
    "Cabinet": 1,
    "ChassisSlots": 0,
    "CustomValues": [],
    "DeviceID": 603,
    "DeviceType": "Server",
    "ESX": 0,
    "EscalationID": 0,
    "EscalationTimeID": 0,
    "FirstPortNum": 0,
    "HalfDepth": 0,
    "Height": 0,
    "InstallDate": "1970-01-01",
    "Label": "STAD9999",
    ...
    ...
    ...
  },
  "error": false,
  "errorcode": 200
}
```

上述代码的说明如下：

- ❑ /device/stad9999 是必需值，device 的 Label。
- ❑ -d Cabinet=1，指定 PUT 方法的要传的参数，即 rack id。

此外，细心的读者可能发现代码中少了以下几个值：

- ❑ CustomValues，这个版本没有 API，只能像 rack 那样通过 SQL 导入，表结构比 rack 还简单，这里不再赘述。
- ❑ Position，位置，如果机器数量少的话，dcim 的 UI 可以直接拖动位置，相当方便，如果机器数量多的话，则必须提前准备好安装的策略，比如靠容量、电力、HA 等，也就是必须事先产生一个包括合理位置信息的 csv，然后再写 Python 导入。

□ TemplateID, 这个可以直接加入, id 和名字的影射关系, 可以参考上文 GET 的例子。

可以看出, 有 API 以后大大减少工作量, 虽然 CustomValues 还不够完美, 但是 open-DCIM 已经有 github issue track 这个问题, 相信不久的将来会和 rack 没有 API 的问题一起解决。

5. API 更新: POST

查询 stad9999 的 DeviceID 如下:

```
[root@opendcim /]# curl -s -u admin:adminpassword '127.0.0.1/api/v1/
device?LABEL=stad9999' | python -m json.tool | grep DeviceID
"DeviceID": 604,
```

查询当前 stad9999 (即 DeviceID:604) 的 Owner 如下:

```
[root@opendcim /]# curl -s -u admin:adminpassword '127.0.0.1/api/v1/
device/604' | python -m json.tool | grep Owner
"Owner": 0,
```

POST 更改如下:

```
[root@opendcim /]# curl -s -u admin:adminpassword -X POST -d Owner=1
'127.0.0.1/api/v1/device/604' | python -m json.tool
{
  "error": false,
  "errorcode": 200
}
```

更改后:

```
[root@opendcim /]# curl -s -u admin:adminpassword '127.0.0.1/api/v1/
device/604' | python -m json.tool | grep Owner
"Owner": 1,
```

上述代码说明如下:

□ 1 次 API 查询得到 STAD9999 的 DeviceID 604。

□ -d Owner=1 用来指定要更改的参数。

□ "error": false, "errorcode": 200, 代表更改成功。

更改的 API 也非常简单, 同样的 CustomValues 和 rack 需要用 SQL 语句更改, 期待新版本的改进。

6. API 删除: DELETE

查询 stad9999 的 DeviceID 如下:

```
[root@opendcim /]# curl -s -u admin:adminpassword '127.0.0.1/api/v1/
device?LABEL=stad9999' | python -m json.tool | grep DeviceID
"DeviceID": 604,
```

删除代码如下：

```
[root@opendcim /]# curl -s -u admin:adminpassword -X DELETE '127.0.0.1/api/v1/device/604'
{"error":true,"errorcode":404,"message":"An unknown error has occured"}
```

通过以下代码确认删除：

```
[root@opendcim /]# curl -s -u admin:adminpassword '127.0.0.1/api/v1/device?LABEL=stad9999' | python -m json.tool
{
  "device": [],
  "error": false,
  "errorcode": 200
}
```

再删除一次，就会报错，如下：

```
sh-4.1# curl -s -u admin:admin -X DELETE '127.0.0.1/api/v1/device/604'
{"error":true,"errorcode":404,"message":"Device doesn't exist"}
```

对上述代码的说明如下：

- ❑ 同样，先查要删除机器的 DeviceID。
- ❑ 在执行删除操作时，系统返回了 404 报错 "An unknown error has occured"。
- ❑ 通过查询确认，发现的确删除了。
- ❑ 再执行一次删除操作，也有“404”的报错，但 message 是 "Device doesn't exist"。

这是 openDCIM 当前不恰当的错误输出判断造成的，目前可以通过检查返回的 message 做判断。必须承认 DELETE 还是有改进的空间的，不过这次 CustomValues 被 API 连带删除了，仅仅 rack 还需要用 SQL 语句更改。

7. 导入一个现实的例子

手工录入表 11-5 中所示的内容。

表 11-5 使用 OpenDCIM 描述业务基础属性

基础属性类别	OpenDCIM 中对应字段	设置参考图例
3 个 Project	Device 默认属性 Departmental Owner	图 11-26 为 Project 录入
3 个 Env	Device 自定义属性 Custom Attributes 中的 env	图 11-27 为 Env 录入
2 个 Dc	使用 opendcim 中的默认 Datacenter 对象	图 11-28 为 Dc 录入
4 个 Server Room	使用 opendcim 中的默认 Zones 对象	图 11-29 为 Server Room 录入
7 种 Device Type	使用 opendcim 中的默认 Device Template 功能	图 11-30 为 Device Type 录入

API 录入表 11-6 所示的内容。

rack 和 server 的导入工作及 host level 的 config 添加在上文已经提到，这里不再赘述，在现实情况中为了方便规划和导入，dc 组常使用 excel 导出 csv，再通过 DevOps 写的 csv

导入工具，从而方便地实现增改工作。

Department: New Department ▼
 Department Name: New Department
 Executive Sponsor: Ecom
 Account Manager: Game
 Department Color: Ops
 Classification: Video
 Internal ▼
 Create

图 11-26 Project 录入

Data Center Configuration

General Workflow Style Email Reporting ToolTips Cabling Custom Device Attributes

Custom Device Attributes

Label	Type	Required	Apply to All Devices	Default Value
env	string	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

图 11-27 env 录入

Data Center Detail

Data Center ID: New Data Center ▼
 Name: New Data Center
 Square Feet: stad
 Delivery Address: tuad
 Administrator:
 Drawing URL:
 Design Maximum (kW):
 Container: None ▼
 X:
 Y:
 Create

图 11-28 DC 录入

Data Center Zones

Zone: New Zone ▼
 Description: New Zone
 Data Center: stad stad-rmb
 X1: stad stad-rmb
 Y1: tuad tuad-rmb
 X2: tuad tuad-rmb
 Y2:
 Zoom (%): 100
 Create

[Return to Main Menu]

图 11-29 Server Room 录入

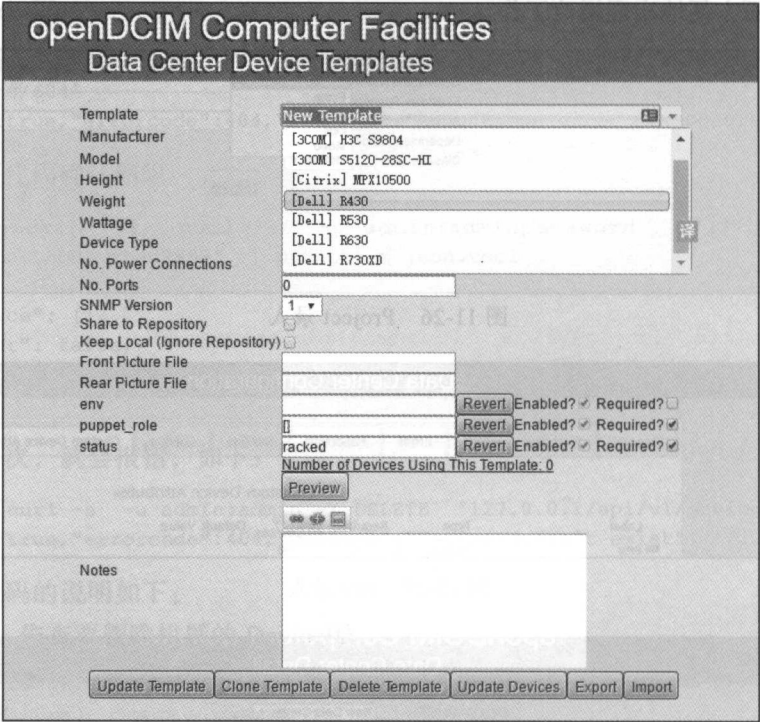


图 11-30 Device Type 录入

表 11-6 使用 openDCIM 描述设备属性

设备属性类型	openDCIM 中的对应字段
80 个 rack	使用 openDCIM 中的默认 Cabinet 对象
500 台 Server+88 台网络设备	使用 openDCIM 中的默认 Device 对象
2 个 Dc_Level+2 个 project+env level 的 conf	使用 openDCIM 中的默认 Device 对象，但命名以前缀 conf_LevelName (conf_stad,conf_game_dev) 来规范，并且以相应的 Departmental Owner, env 和 Datacenter 来关联相应的 host。之后 conf 管理，使用 Device 自定义属性 Custom Attribute 的功能
若干 host level 的 conf	Device 自定义属性 Custom Attribute 中的 env

csv 读取的 Python 代码如下：

```
#!/usr/bin/python2.7
import csv
def read_csv(dcm_csv):
    with open(dcm_csv, mode='r') as csvfile:
        reader = csv.DictReader(csvfile)
        dcm_dict = {}
        for row in reader:
            dcm_dict[row['Label']] = row
    return dcm_dict
```

csv 的样式见图 11-31。

```
Label,Cabinet,TemplateID,Owner,env,status,puppet_role
STAD-RMA-CSU01,stad-rna-ru01-rk01,H3C S9804,Ops,prod,live,[]
STAD-RMA-CSU02,stad-rna-ru02-rk01,H3C S9804,Ops,prod,live,[]
STAD-RMB-CSU01,stad-rnb-ru01-rk01,H3C S9804,Ops,prod,live,[]
STAD-RMB-CSU02,stad-rnb-ru02-rk01,H3C S9804,Ops,prod,live,[]
TUAD-RMA-CSU01,tuad-rna-ru01-rk01,H3C S9804,Ops,prod,live,[]
TUAD-RMA-CSU02,tuad-rna-ru02-rk01,H3C S9804,Ops,prod,live,[]
TUAD-RMB-CSU01,tuad-rnb-ru01-rk01,H3C S9804,Ops,prod,live,[]
TUAD-RMB-CSU02,tuad-rnb-ru02-rk01,H3C S9804,Ops,prod,live,[]
STAD-RMA-LB01,stad-rna-ru01-rk01,MPX10500,Ops,prod,live,[]
STAD-RMA-LB02,stad-rna-ru02-rk01,MPX10500,Ops,prod,live,[]
STAD-RMB-LB01,stad-rnb-ru01-rk01,MPX10500,Ops,prod,live,[]
STAD-RMB-LB02,stad-rnb-ru02-rk01,MPX10500,Ops,prod,live,[]
TUAD-RMA-LB01,tuad-rna-ru01-rk01,MPX10500,Ops,prod,live,[]
TUAD-RMA-LB02,tuad-rna-ru02-rk01,MPX10500,Ops,prod,live,[]
TUAD-RMB-LB01,tuad-rnb-ru01-rk01,MPX10500,Ops,prod,live,[]
TUAD-RMB-LB02,tuad-rnb-ru02-rk01,MPX10500,Ops,prod,live,[]
STAD-RMA-RU01,stad-rna-ru01-rk01,S512R-28SC-41,Ops,prod,[]
```

图 11-31

图 11-31 中的 csv 文件以及读取该文件的 Python 代码说明如下：

- read_csv 返回一个以 label 为 key 的字典，方便其他函数调用。
- env、status、puppet_role 为 Custom Attribute，需要等第一步 device API 导入后，再 SQL 关联相应设备后导入。

下面将着重介绍 dc_level 及 Project+Env_Level 的 Conf 实现方法。

1) 添加虚拟 dc，“conf”，用作包装不同级别专属变量的上层容器，如“dc_level”、“project_env_level”。

2) 添加虚拟机柜，“dc_level”、“project_env_level”，用作包装不同 dc 和不同 project 专属变量的上层容器，如“STAD_CONF”、“ECOM_DEV_CONF”。

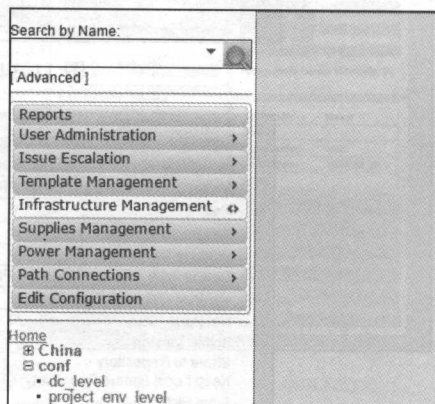


图 11-32 虚拟 dc/ 机柜创建

前两步效果如图 11-32 所示。

3) 创建 Customer Device Attributes，如图 11-33 所示。

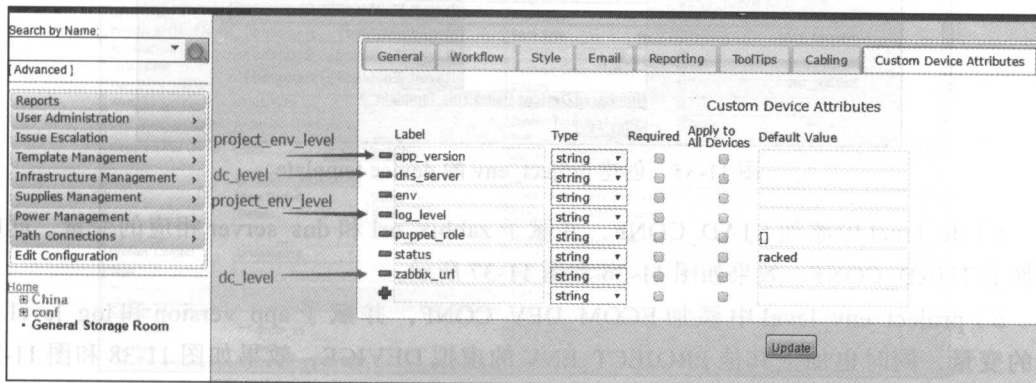


图 11-33 dc_level/project_env_level 的自定义属性创建

4) 创建 dc 的 device template 和 project_env 的 device template，如图 11-34 和图 11-35 所示，注意勾选需要的自定义属性。

The screenshot shows the configuration for a device template named 'DC_CONF'. The interface includes a list of attributes on the left and their corresponding values on the right. The 'app_version', 'dns_server', 'env', 'log_level', 'puppet_role', 'status', and 'zabbix_url' attributes have associated 'Revert', 'Enabled?', and 'Required?' checkboxes. The 'Required?' checkboxes for 'dns_server', 'env', 'log_level', and 'zabbix_url' are checked. The 'Number of Devices Using This Template' is 2.

Attribute	Value	Revert	Enabled?	Required?
Template	[conf] DC_CONF			
Manufacturer	conf			
Model	DC_CONF			
Height	0			
Weight	0			
Wattage	0			
Device Type	Physical Infrastructure			
No. Power Connections	0			
No. Ports	0			
SNMP Version	1			
Share to Repository	<input type="checkbox"/>			
Keep Local (Ignore Repository)	<input type="checkbox"/>			
Front Picture File				
Rear Picture File				
app_version		Revert	Enabled?	Required? <input checked="" type="checkbox"/>
dns_server		Revert	Enabled? <input checked="" type="checkbox"/>	Required? <input checked="" type="checkbox"/>
env		Revert	Enabled? <input checked="" type="checkbox"/>	Required? <input checked="" type="checkbox"/>
log_level		Revert	Enabled? <input checked="" type="checkbox"/>	Required? <input checked="" type="checkbox"/>
puppet_role	0	Revert	Enabled?	Required?
status	racked	Revert	Enabled?	Required?
zabbix_url		Revert	Enabled? <input checked="" type="checkbox"/>	Required? <input checked="" type="checkbox"/>

Number of Devices Using This Template: 2

Preview

图 11-34 创建 dc 的 device template

The screenshot shows the configuration for a device template named 'PROJECT_ENV_CONF'. The interface includes a list of attributes on the left and their corresponding values on the right. The 'app_version', 'dns_server', 'env', 'log_level', 'puppet_role', 'status', and 'zabbix_url' attributes have associated 'Revert', 'Enabled?', and 'Required?' checkboxes. The 'Required?' checkboxes for 'app_version', 'env', and 'log_level' are checked. The 'Number of Devices Using This Template' is 9.

Attribute	Value	Revert	Enabled?	Required?
Template	[conf] PROJECT_ENV_CONF			
Manufacturer	conf			
Model	PROJECT_ENV_CONF			
Height	0			
Weight	0			
Wattage	0			
Device Type	Physical Infrastructure			
No. Power Connections	0			
No. Ports	0			
SNMP Version	1			
Share to Repository	<input type="checkbox"/>			
Keep Local (Ignore Repository)	<input type="checkbox"/>			
Front Picture File				
Rear Picture File				
app_version		Revert	Enabled? <input checked="" type="checkbox"/>	Required? <input checked="" type="checkbox"/>
dns_server		Revert	Enabled?	Required?
env		Revert	Enabled? <input checked="" type="checkbox"/>	Required? <input checked="" type="checkbox"/>
log_level		Revert	Enabled? <input checked="" type="checkbox"/>	Required? <input checked="" type="checkbox"/>
puppet_role	0	Revert	Enabled?	Required?
status	racked	Revert	Enabled?	Required?
zabbix_url		Revert	Enabled?	Required?

Number of Devices Using This Template: 9

Preview

图 11-35 创建 project_env 的 device template

5) dc_level 中添加 STAD_CONF，并赋予 zabbix_url 和 dns_server 相应的变量，同时也加上 TUAD_CONF，效果如图 11-36 和图 11-37 所示。

6) project_env_level 中添加 ECOM_DEV_CONF，并赋予 app_version 和 log_level 相应的变量，同时也加上其他 PROJECT_ENV 的虚拟 DEVICE，效果如图 11-38 和图 11-39 所示。

Asset Tracking

Device ID: 599
 Reservation? ☐
 Label: STAD_CONF
 Serial Number:
 Asset Tag:
 Primary IP / Host Name:
 Manufacture Date: 01/01/1970
 Install Date: 05/24/2016
 Warranty Company:
 Warranty Expiration: 01/01/1970
 Last Audit Completed: Audit not yet completed
 Departmental Owner: Ops

Escalation Information

Time Period: Select...
 Details: Select...

Primary Contact: Unassigned
 Tags: Awaiting input...

Custom Attributes

zabbix_url: 1.1.1.10
 dns_server: 1.1.1.11

Physical Infrastructure

Cabinet: conf / dc_level
 Device Class: conf - DC_CONF
 Height: 0
 Position: 0
 Half Depth: ☐
 Back Side: ☐
 Number of Data Ports: 0
 Nominal Draw (Watts): 0
 Weight: 0
 Power Connections: 0
 Device Type: Physical Infrastructure

SNMP Configuration

SNMP Version: 2c
 SNMP Read Only Community:
 Consecutive SNMP Failures: 0
 *Polling is disabled after three consecutive failures.

图 11-36 创建 device STAD_CONF

Search by Name:

Advanced

Reports

- User Administration
- Issue Escalation
- Template Management
- Infrastructure Management
- Supplies Management
- Power Management
- Path Connections
- Edit Configuration

Home

- China
- IS conf
- dc_level
- project env_level
- Storage Room
- General Storage Room

Images / Labels

dc_level dc_level (Rear)

Pos Device Pos Device

Markup Key

Zero-U Devices
 STAD_CONF
 TUDL_CONF

Power Distribution

Add CDU

Environmental Sensors

Add Sensor

Cabinet Metrics

Space: 0%

Weight (Maximum Weight Not Set)

Computed Watts: 0 kW / 0 kW

Measured Watts: 0 kW / 0 kW

Approximate Center of Gravity: 0U

Last Audit: Never

Model:
 Data Center: conf
 Install Date: 2016-05-23

Tags:

图 11-37 dc_level 整体效果

Asset Tracking

Device ID: 625
 Reservation? ☐
 Label: ECOM_DEV_CONF
 Serial Number:
 Asset Tag:
 Primary IP / Host Name:
 Manufacture Date: 01/01/1970
 Install Date: 01/01/1970
 Warranty Company:
 Warranty Expiration: 01/01/1970
 Last Audit Completed: Audit not yet completed
 Departmental Owner: Ops

Escalation Information

Time Period: Select...
 Details: Select...

Primary Contact: Unassigned
 Tags: Awaiting input...

Custom Attributes

app_version: ecom_v3
 log_level: debug

Physical Infrastructure

Cabinet: conf / project env_level
 Device Class: conf - PROJECT_ENV_CONF
 Height: 0
 Position: 0
 Half Depth: ☐
 Back Side: ☐
 Number of Data Ports: 0
 Nominal Draw (Watts): 0
 Weight: 0
 Power Connections: 0
 Device Type: Server

SNMP Configuration

SNMP Version: 2c
 SNMP Read Only Community:
 Consecutive SNMP Failures: 0
 *Polling is disabled after three consecutive failures.

VMWare ESX Server Information

ESX Server? False

图 11-38 创建 device 加 ECOM_DEV_CONF

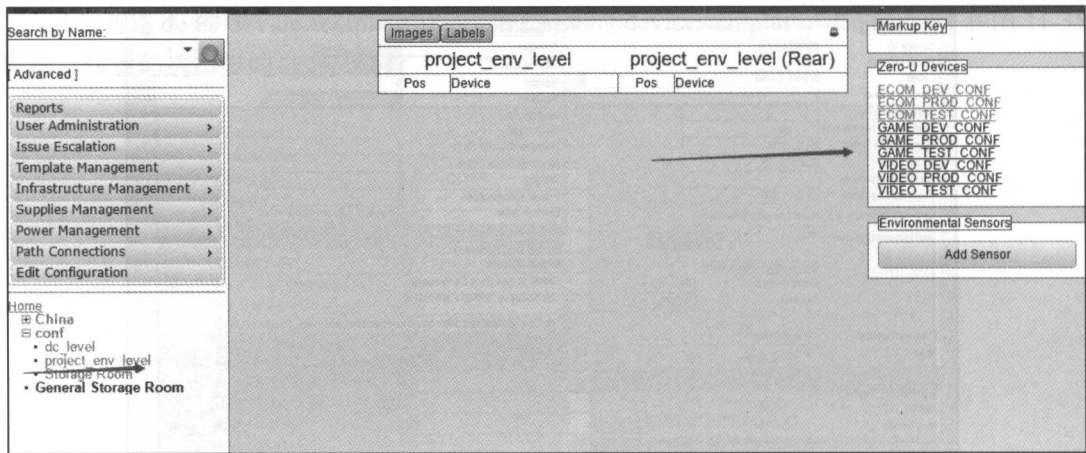


图 11-39 project_env_level 整体效果

细心的读者会发现，以上加的都是 device 资源，因此我们完全可以利用 API 进行增删改查操作，一个简单的 shell 范例如下：

```
[root@opendcim /]# for p in game video ecom; do for e in dev test prod; do
    curl -s -u admin:adminpassword -X PUT -d Cabinet=85 -d Owner=1 -d
    TemplateID=10 '127.0.0.1/api/v1/device/'
    $p"_"$e"_conf" | python -m json.tool;done;done
```

上述代码说明如下：

- ❑ 2 层循环，迭代 9 次。
- ❑ 相应的 ID 是从其他资源的 API 里查出的。

下面是查询的示例：

```
[root@opendcim /]# curl -s -u admin:adminpassword '127.0.0.1/api/v1/
device?Label=stad_conf' | python -m json.tool| grep -E 'zabbix|dns'
    "dns_server": "1.1.1.11",
    "zabbix_url": "1.1.1.10"
[root@opendcim /]# curl -s -u admin:adminpassword '127.0.0.1/api/v1/
device?Label=ecom_dev_conf' | python -m json.tool| grep -E 'app|log_level'
    "app_version": "ecom_v3",
    "log_level": "debug",
```

在上述代码中，Label 就是 Device 名字的属性名，也就是刚才创建的那些虚拟资源名。就是这样一个简单的 API，是不是非常方便？接下去就可以完全接入 Puppet 变成真正意义上的 CMDB 中心。

综上所述，openDCIM 的 API，加上略微的定制，基本可以完全胜任 CMDB 的工作，API 也将在未来版本趋于完美，下一节将结合 11.4.1 节，将 openDCIM 的运用进一步扩展开来。

11.5.4 解决每个项目都会遇到的那些任务

1. ops 和 dev 一起评估所需新机器数量

上文提到过，空闲机器数 / 现有机器数及角色是这一步主要关心的 2 个 CMDB 信息，openDCIM 完全可以从 API 满足这一需求，上文也有 API 的详解，这类不再赘述。而对于 UI 方式，下面给出一个更加便捷的查询方式。

通过图 11-40 中所示的 DC 信息和 Export 功能和图 11-41 所示的 DC 信息和 Export 表，可以直观地看到所有的信息，如哪些 DC，包含哪些机柜，分别放置了哪些项目的服务器。

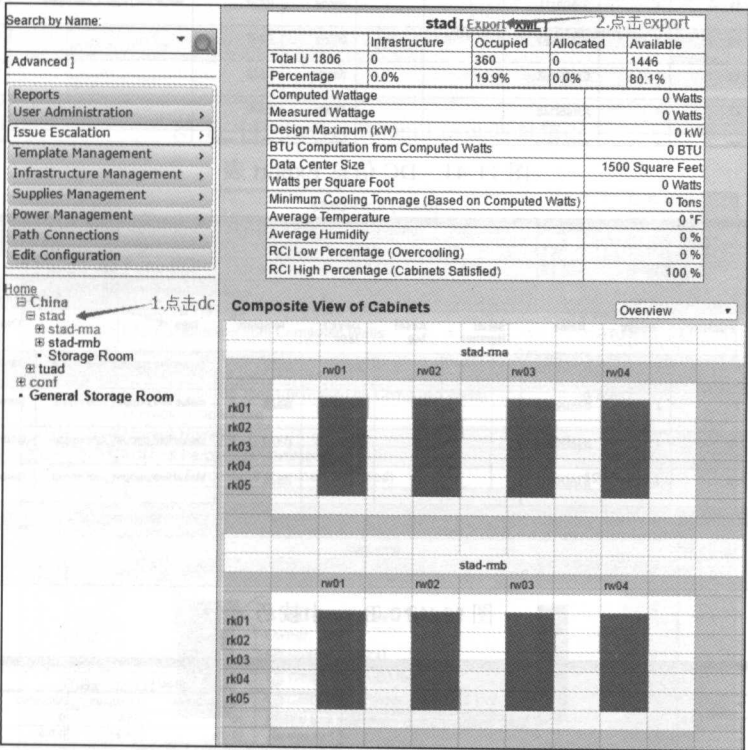


图 11-40 DC 信息 Export 功能

同时 search 的功能可以过滤出想查看的信息，这里用了 tags 的小技巧，原因是目前 openDCIM 对于 custom attribute 的 value 支持还不够完美，因此复制其 key、value 到 tags 是一个比较好的技巧，这样可以完全满足现实需要（如图 11-42 所示）。

2. 评估机房容量，从电力，网络设备容量，机柜空闲等多个维度

容量评估是 openDCIM 做得比较出彩的地方，以下几张图可以让一个 dc 管理人员自上而下的观察到所有容量情况。

第一步，通过图 11-43 所示的图看 dc level 容量情况，目前因为都是虚拟出来的设备，所有没有电力数据，有兴趣的读者可以通过 ipmitool 取得机器的电力数据。

Data Center View/Export

admin/4.1

Data Center: Select data center ▼

Copy CSV Excel Print Show / hide columns

Show 25 ▼ entries

Search:

Data Center	Location	Position	Height	Name	Serial Number	Asset Tag	Device Type	Template	Tags	Owner	Installation Date
stad	stad-rma-rw01-rk01	39	2	STAD-RMA-CSW01			Server	H3C-S9804		Ops	26 Jul 2015
stad	stad-rma-rw01-rk02	1	2	STAD0001			Server	R730XD		Ops	26 Jul 2015
stad	stad-rma-rw01-rk02	4	1	STAD0013			Server	R630	status:live,puppet_role:mysql	Game	26 Jul 2015
stad	stad-rma-rw01-rk02	7	2	STAD0045			Server	R530	status:live,puppet_role:mysql	Game	26 Jul 2015
stad	stad-rma-rw01-rk02	10	1	STAD0093			Server	R430	status:live,puppet_role:mysql	Game	26 Jul 2015
stad	stad-rma-rw01-rk02	13	1	STAD0111			Server	R430	status:live,puppet_role:mysql	Game	27 Jul 2015
stad	stad-rma-rw01-rk02	16	2	STAD0063			Server	R530		Video	27 Jul 2015
stad	stad-rma-rw01-rk02	19	1	STAD0147			Server	R430		Ecom	27 Jul 2015
stad	stad-rma-rw01-rk02	22	1	STAD0129			Server	R430		Game	27 Jul 2015
stad	stad-rma-rw01-rk02	25	1	STAD0111			Server	R430		Game	27 Jul 2015

图 11-41 DC 信息 Export 表

Search: Select data center ▼

Copy CSV Excel Print Show / hide columns

Show 25 ▼ entries

Search: mysql

复制 Custom Attributes 到 Tags

Data Center	Location	Position	Height	Name	Serial Number	Asset Tag	Device Type	Template	Tags	Owner	Installation Date
stad	stad-rma-rw01-rk02	4	1	STAD0013			Server	R630	status:live,puppet_role:mysql	Game	26 Jul 2015
stad	stad-rma-rw01-rk02	7	2	STAD0045			Server	R530	status:live,puppet_role:mysql	Game	26 Jul 2015
stad	stad-rma-rw01-rk02	10	1	STAD0093			Server	R430	status:live,puppet_role:mysql	Game	26 Jul 2015
stad	stad-rma-rw01-rk02	13	1	STAD0111			Server	R430	status:live,puppet_role:mysql	Game	27 Jul 2015

Showing 1 to 4 of 4 entries (filtered from 293 total entries)

Previous 1 Next

4条记录

图 11-42 Tags 小技巧

Search by Name:

[Advanced]

Reports

User Administration

Issue Escalation

Template Management

Infrastructure Management

Supplies Management

Power Management

Path Connections

Edit Configuration

Home

China

stad

stad-rma

stad-rmb

Storage Room

第一步，点击DC

机房总剩余U数

机房总电力情况

stad (Export XML)

Infrastructure	Occupied	Allocated	Available
Total U 1806	0	360	0
Percentage	0.0%	19.9%	0.0%
Computed Wattage			0 Watts
Measured Wattage			0 Watts
Design Maximum (kW)			0 kW
BTU Computation from Computed Watts			0 BTU
Data Center Size			1500 Square Feet
Watts per Square Foot			0 Watts
Minimum Cooling Tonnage (Based on Computed Watts)			0 Tons
Average Temperature			0 °F
Average Humidity			0 %
RC1 Low Percentage (Overcooling)			0 %
RC1 High Percentage (Cabinets Satisfied)			100 %

Composite View of Cabinets

Overview

stad-rma

图 11-43 dc level 容量情况

第二步，通过图 11-44 所示的图看 Server Room Level 容量情况，和 dc level 视图一致。
第三步，通过图 11-45 所示的图看 Rack Level 容量情况，鼠标移到相应格子，即会弹

出相应 rack 的数据，点击即可进入相应 rack，看下一层的数据。

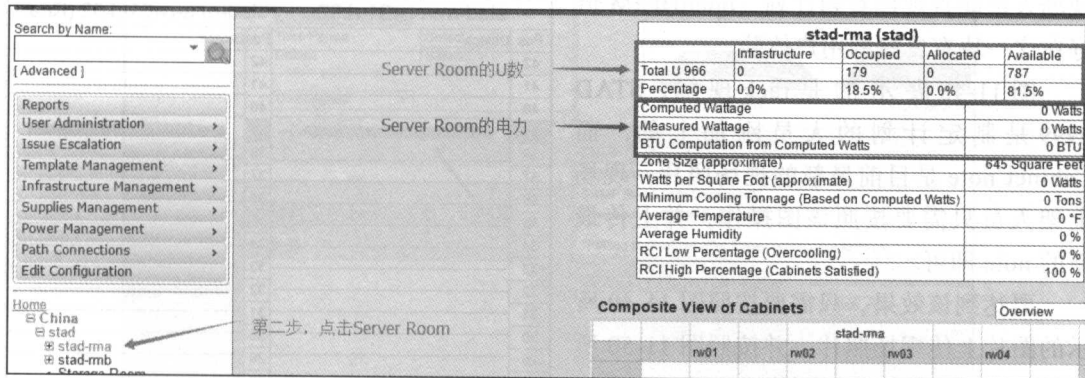


图 11-44 Server Room Level 容量情况

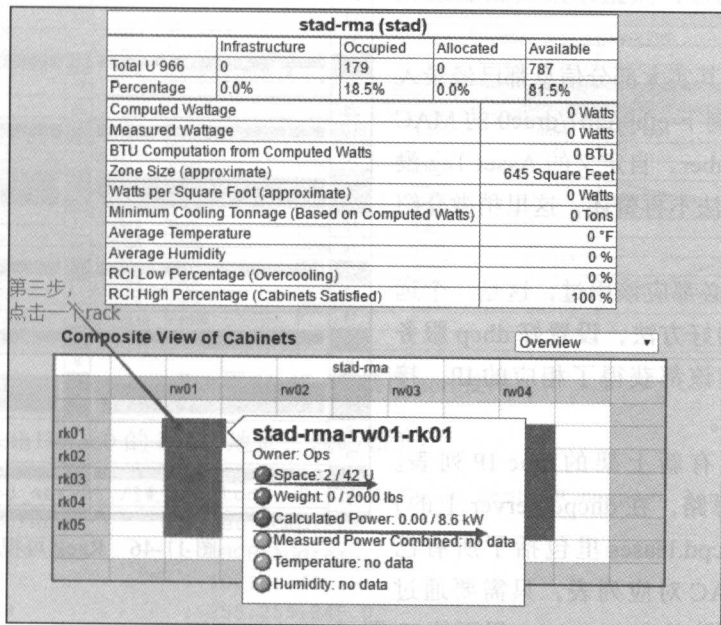


图 11-45 Server Room Level 容量情况

第四步，通过图 11-46 所示的图看 rack 可视图的前面和背面，点击该 rack 的交换机。
 第五步，图 11-47 显示了交换机 port 的使用情况，从而得知交换机的 port 使用情况。
 至此，DC 管理人员完全可了解是否有足够的容量供新进一批机器的上架了。

3. 制定扩容方案，包括新机器放置位置，交换机连线和角色分配

如果机器数量少的话，可以在 UI 上直接添加新的 device，并把 status 设置为 unrack。
 如果数量多的话，目前只能导出现有数据成 csv，通过 excel 编辑，同样可把新机器 status

好在这一块有良好的用户体验。

新的 note 即可。

示的图片设置机柜现场照片。

连线

一下收集方法。

下去的步骤如下。

opendcim API 剔除已在 opendcim 里面的 IP

2) 得到 eth0/eth1/drac0 的 MAC 地址, serial number。

根据 user id 改 vendor 默认密码，代码如下：

```
[root@dhcpd_server /]# ipmitool -H <IPADDR> -U root -P changeme user list
```

ID	Name	Callin	Link Auth	IPMI Msg	Channel Priv Limit
1		true	true	true	NO ACCESS
2	root	true	true	true	ADMINISTRATOR

```
[root@dhcpd_server /]# ipmitool -H <IPADDR> -U root -P changeme user set password
2 newpass
```

stad-rma-rw01-rk02		stad-rma-rw01-rk02 (Rear)	
Pos	Device	Pos	Device
42		42	
41		41	
40		40	
39	STAD-RMA-RW01-RK02-ASW	39	STAD-RMA-RW01-RK02-ASW
38		38	
37		37	
36		36	
35		35	
34		34	
33		33	
32		32	
31		31	
30		30	
29		29	
28		28	
27		27	
26		26	
25		25	
24		24	
23		23	
22	STAD0129	22	STAD0129(Rear)
21		21	
20		20	
19	STAD0147	19	STAD0147(Rear)
18		18	
17	STAD0063	17	STAD0063(Rear)
16		16	
15		15	
14		14	
13	STAD0111	13	STAD0111(Rear)
12		12	
11		11	
10	STAD0093	10	STAD0093(Rear)
9		9	
8	STAD0045	8	STAD0045(Rear)
7		7	
6		6	
5		5	
4	STAD0013	4	STAD0013(Rear)
3		3	

图 11-46 Rack 可视图

Last Audit Completed: Audit not yet completed

Departmental Owner: Ops | Show Contacts

Escalation Information: Time Period: Select... Details: Select...

Primary Contact: Unassigned | Tags: Awaiting input...

Custom Attributes: Preview

Notes

Device type: Server

Device Images: S5120-28SC-HI, S5120-28SC-HI

SNMP Configuration: SNMP Version: 2c, SNMP Read Only Community: , Consecutive SNMP Failures*: 0, *Polling is disabled after three consecutive failures.

VMWare ESX Server Information: ESX Server?: False

Operational Log: Date:

Power Connections: # | Port Name | Device | Device Port

1 | Power Connection 1 | |

2 | Power Connection 2 | |

Limit device selection to: Row | Zone | Data

Connections: # | Port Name | Device | Device Port | Notes | Media Type | Color C

1 | Port1 | | | | |

2 | Port2 | STAD0129 | Port1 | | |

3 | Port3 | STAD0147 | Port1 | | |

4 | Port4 | STAD0063 | Port1 | | |

5 | Port5 | STAD0111 | Port1 | | |

6 | Port6 | STAD0093 | Port1 | | |

7 | Port7 | STAD0045 | Port1 | | |

8 | Port8 | STAD0013 | Port1 | | |

9 | Port9 | STAD0001 | Port1 | | |

10 | Port10 | | | | |

11 | Port11 | | | | |

12 | Port12 | | | | |

图 11-47 交换机 port 使用

获取 eth0/eth1/drac0 的 MAC 地址，代码如下：

```
[root@dhcpd_server /]# ipmitool -H <IPADDR> -U root -P newpass lan print
Set in Progress      : Set Complete
IP Address Source    : DHCP Address
IP Address           : 10.191.40.13
Subnet Mask          : 255.255.248.0
MAC Address          : e0:24:7f:b7:a4:10
```

上述代码的说明如下：

- ❑ ipmi 只能或者 drac 的 MAC 地址，但是 Intel 平台的板载网卡，基本都是：
eth0 mac address+1=eth1 mac address
eth1 mac address+1=drac mac address
而且在下一步中的 mini ISO，会有一步验证工作。
 - ❑ 大部分 vendor 可在出厂时预先提供 MAC 地址，这样可以省去部分麻烦。
- 获取 serial number 的代码如下：

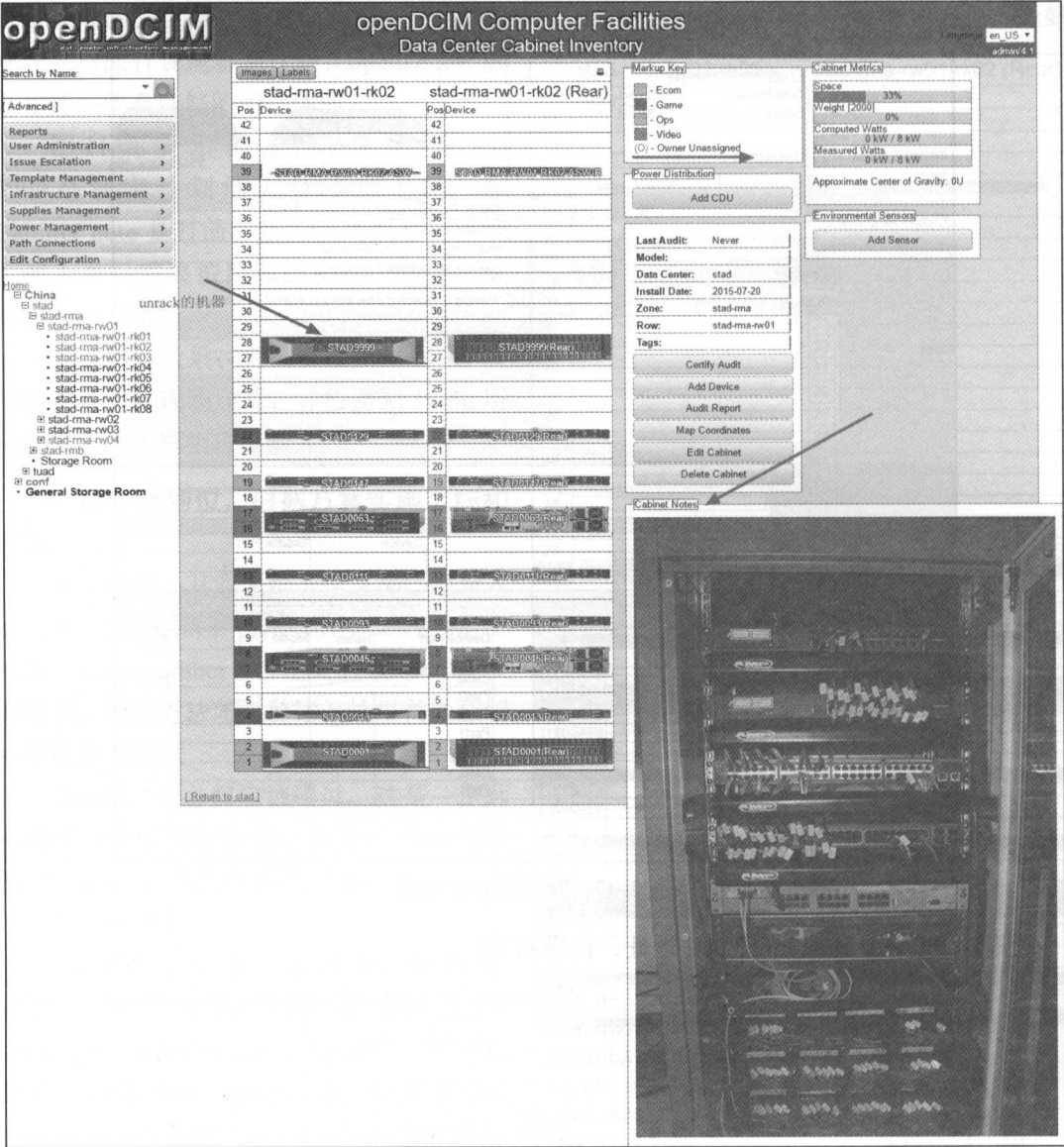


图 11-48 rack 现场

```
[root@dhcpd_server /]# ipmitool -H <IPADDR> -U root -P newpass fru
FRU Device Description : Builtin FRU Device (ID 0)
Board Mfg Date        : Sat May 12 10:42:00 2012
Board Mfg              : Huawei Technologies Co., Ltd.
Board Product         : XX21XXXX0
Board Serial          : 030XXX10X5000027
Product Manufacturer: Huawei Technologies Co., Ltd.
Product Serial        : 2XXXXXXXXXXXXXXXXX-3
Product Ass t Tag     :
```

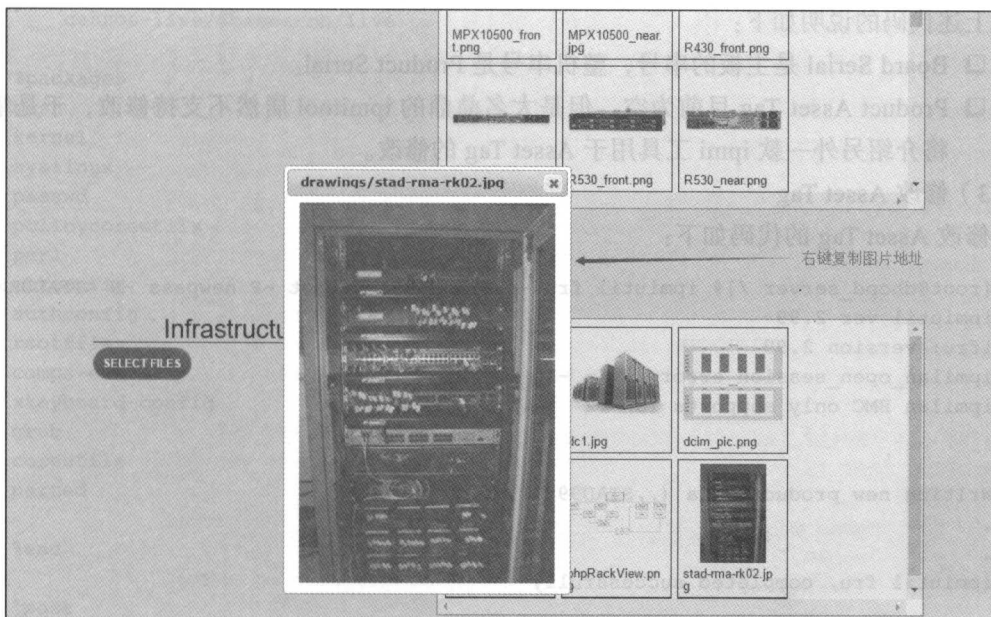


图 11-49 上传现场照片

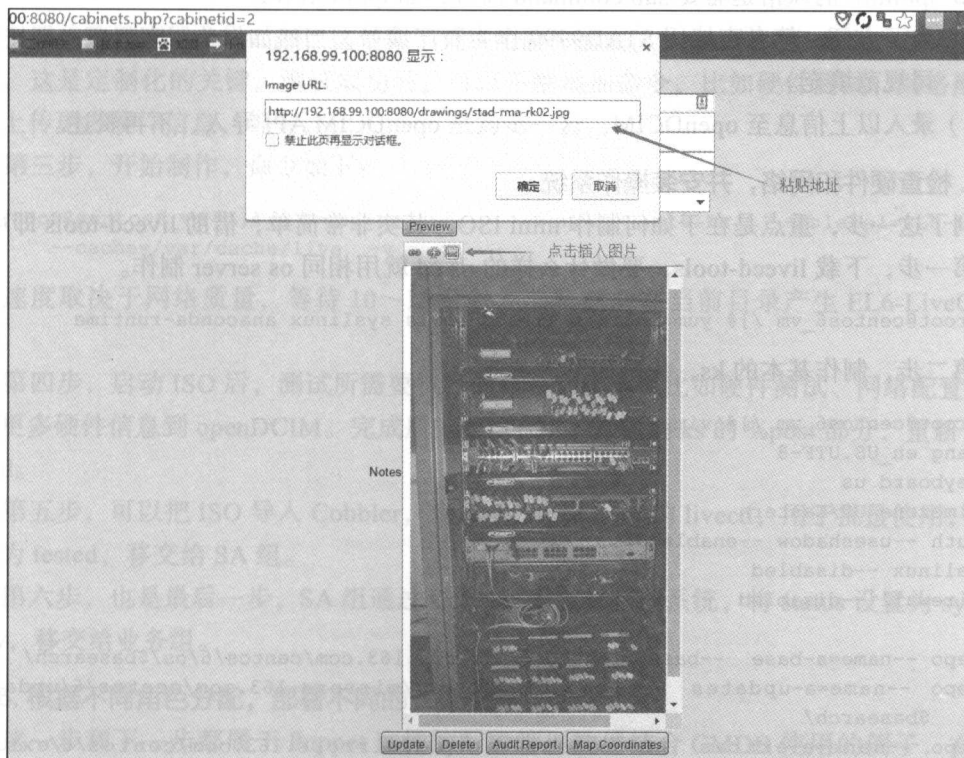


图 11-50 设置机柜现场照片

上述代码的说明如下：

- ❑ Board Serial 是主板的串号，整机串号是 Product Serial。
- ❑ Product Asset Tag 目前为空，但是大名鼎鼎的 ipmitool 居然不支持修改，于是笔者将介绍另外一款 ipmi 工具用于 Asset Tag 的修改。

3) 修改 Asset Tag

修改 Asset Tag 的代码如下：

```
[root@dhcpd_server /]# ipmiutil fru -a STAD9999 -U root -P newpass -N <IPADDR>
ipmiutil ver 2.99
ifru: version 2.99
ipmilan_open_session error, rv = -15
ipmilan BMC only supports lan v2
.
.
Writing new product data (,,STAD9999) ...
.
.
ipmiutil fru, completed successfully
```

上述代码的说明如下：

- ❑ ipmiutil 的风格是需要 sub command 前置，如 ipmiutil fru。
- ❑ 为了方便，笔者直接用 STAD9999 作为资产编号，当然如果公司有规范，即按照公司规范填充。

4) 录入以上信息至 openDCIM，这一步就是 openDCIM API 导入，不再赘述。

5. 检查硬件和网络，并安装操作系统

到了这一步，重点是在于如何制作 mini ISO，其实非常简单，借助 livecd-tools 即可。第一步，下载 livecd-tools，要做什么样的 os cd 就用相同 os server 制作。

```
[root@centos6_vm /]# yum install livecd-tools syslinux anaconda-runtime
```

第二步，制作基本的 ks，代码如下：

```
[root@centos6_vm /]# vim el6.ks
lang en_US.UTF-8
keyboard us
timezone US/Eastern
auth --useshadow --enablemd5
selinux --disabled
firewall --disabled

repo --name=a-base --baseurl=http://mirrors.163.com/centos/6/os/$basearch/
repo --name=a-updates --baseurl=http://mirrors.163.com/centos/6/updates/
    $basearch/
repo --name=a-extras --baseurl=http://mirrors.163.com/centos/6/extras/
    $basearch/
repo --name=a-live --baseurl=http://www.nanotechnologies.qc.ca/propos/linux/
```

```
centos-live/$basearch/live

%packages
bash
kernel
syslinux
passwd
policycoreutils
perl
chkconfig
authconfig
rootfiles
comps-extras
xkeyboard-config
grub
coreutils
parted

%end

%post
echo testing >/opt/livecd.log
%end
```

上述代码使用了 163 repo，这是为了更快的制作速度。%post 就一条命令，是为了方便测试，这是定制化的关键，测试成功后，可以不断地加命令，比如硬件测试、网络配置验证、上传更多硬件信息，等等。

第三步，开始制作，命令如下：

```
[root@centos6_vm /]# livecd-creator --config=./el6.ks --fslabel=EL6-LiveCD
--cache=/var/cache/live -v
```

速度取决于网络质量，等待 10~15 分钟左右，就会在当前目录产生 EL6-LiveCD.iso 文件。

第四步，启动 ISO 后，测试所需要执行的命令和脚本，比如硬件测试、网络配置验证，上传更多硬件信息到 openDCIM。完成后把 HISTORY 打包到 ks 的 %post 部分，重新 create livecd。

第五步，可以把 ISO 导入 Cobbler，变成基于 pxe boot 的 livecd，用于批量使用，status 设置为 tested，移交给 SA 组。

第六步，也是最后一步，SA 组通过 Cobbler 安装好基础系统，将 status 设置为 system_ready，移交给业务组。

6. 根据不同角色分配，部署不同的业务软件

这一步和下一步都属于 Puppet 这样的配置管理软件结合 CMDB 使用的例子，在下一步中，将给出 openDCIM 结合 Puppet 的实例。

7. 测试完成后，上线，接入各种系统，如 LB、监控

在 Puppet 章节中，笔者介绍过 enc 的配置方法，以及 enc 脚本输出的正确格式（需满足 Puppet 的官方规范），那么 openDCIM 相关的 enc 脚本，只需要满足这个 format，即可接入 Puppet 使用，代码如下：

```
[root@puppet_master /]# cat /usr/local/bin/opendcim_enc.py
#!/usr/bin/python
import requests
import sys
import re
import yaml
import ast

# NO api in 4.2, will be fixed in 4.3
MAPPING_DEPT_ID = {
    "1": "ops",
    "2": "game",
    "3": "ecom",
    "4": "video",
}

class GetParameters(object):
    def __init__(self, outcome, host, url='http://127.0.0.1/api/v1', username='admin', password='admin'):
        self.host = host.lower()
        self.outcome = outcome

        self.url = url
        self.req = requests.Session()
        self.req.auth = (username, password)

        self.host_data = self.get('/device?label=%s' % self.host)
        self.host_data['Datacenter'] = re.sub('[0-9]*', '', self.host)

    def __check_return(self, ret, entry):
        if ret.status_code != 200:
            sys.exit("Error: %s" % ret.text)
        else:
            entry_r = re.sub('\?.*', '', entry)
            ret_key = entry_r.replace("/", "")
            if re.search('\?', entry):
                if len(ret.json()[ret_key]) > 1:
                    sys.exit('multiple entries with %s, fix inventory first' % entry)
                elif len(ret.json()[ret_key]) == 0:
                    sys.exit('no entries with %s, check your inventory' % entry)
                else:
                    return ret.json()[ret_key][0]
            else:
                return ret.json()[ret_key]
```

```

def get(self, entry, filter=[]):
    url = self.url + entry
    r = self.req.get(url)
    checked_return = self.__check_return(r, entry)
    if filter:
        new_return = []
        for i in checked_return:
            # datacenter entry return a list
            if isinstance(checked_return, list):
                one_entry_attr = i
            # other entry return a dict
            elif isinstance(checked_return, dict):
                one_entry_attr = checked_return[i]

            new_element = { k: one_entry_attr[k] for k in filter }
            new_return.append(new_element)

        return new_return
    else:
        return checked_return

def __get_dc_params(self):
    dc_data = self.get('/device?label=%s_conf' % self.host_data['Data-center'])
    dc_conf = dc_data.copy()
    for k in dc_data:
        if k in self.host_data:
            del dc_conf[k]
    self.outcome["parameters"].update(dc_conf)

def __get_project_env_params(self):
    project_id = self.host_data['Owner']
    self.project = MAPPING_DEPT_ID[str(project_id)]
    self.host_data['project'] = self.project
    if self.project != 'ops' and 'env' in self.host_data:
        project_env_data = self.get('/device?label=%s_%s_conf' % (self.project, self.host_data['env']))
        project_env_conf = project_env_data.copy()
        for k in project_env_data:
            if k in self.host_data:
                del project_env_conf[k]
        self.outcome["parameters"].update(project_env_conf)

def __get_host_params(self):
    self.outcome["parameters"].update(self.host_data)
    self.outcome["classes"].update({self.host_data['puppet_role']: {}})
    self.outcome["environment"] = self.host_data['env']

def run(self):
    self.__get_dc_params()
    self.__get_project_env_params()

```



```

        self.__get_host_params()
        return self.outcome

def main(host):
    default_outcome = {"classes": {"puppet-agent": {}}, "parameters": {}, "environment": "test"}
    final_outcome = GetParameters(default_outcome, host).run()
    print yaml.safe_dump(final_outcome, default_flow_style=False)

if __name__ == "__main__":
    try:
        host = sys.argv[1]
    except:
        print "I need a hostname as sys.argv[1]"
    main(host)

```

这是一个非常粗糙的 enc 脚本，旨在实现说明，并不考虑 timeout、log 等情况。可以看出主体在用 GetParameters 这个类的 run()，而 run() 就三步：self.__get_dc_params()、self.__get_project_env_params()、self.__get_host_params()，依层次迭代 self.outcome 这个对象。

最终可以达到如下效果：

```

[root@puppet_master /]# /usr/local/bin/opendcim_enc.py stad0001
classes:
  puppet-agent: {}
  zabbix_server: {}
environment: prod
parameters:
  AssetTag: 'xxx'
  Cabinet: 2
  Datacenter: stad
  .
  .
  .
  Label: STAD0001
  app_version: 1.1.22
  dns_server: 1.1.1.11
  env: prod
  project: game
  puppet_role: zabbix_server
  status: live
  zabbix_url: 1.1.1.10
  .
  .
  .

```

至此，一个“每个项目都会遇到的那些任务”已经基本实现，虽然由于篇幅和每个项目的特殊性，具体步骤无法在本书里详细给出，但是大体思路已经一一给出，读者可以根据自己项目实际，指定实施计划，毕竟真正在一个项目上 CMDB，本身就是需要长期持续

化地推动，不是一本书可以讲得完的。笔者项目也在不断地探索最佳实践，也希望能和行业里的攻城狮多多交流，一起探索这充满挑战的技术海洋！

11.6 如何管理好一个 CMDB

11.6.1 制定相应流程管理

提到流程管理，不得不重申大名鼎鼎的 ITIL，很多人对它懵懵懂懂，比如初露锋芒的技术少年，很多人对它深恶痛绝，比如大公司的螺丝钉们，也有很多人对它深信不疑，当然他们大多数是老板。无论哪种人，都应该认同 ITIL 的目标，做好 change management，避免错误的更改，而 CMDB 作为 Puppet 这样的自动化配置工具的信息源，就更需要保证攻城狮在更改信息时的正确性，否则，后果便是线上服务器错误配置造成的直接事故。因此，流程管理对于管理 CMDB 来说，是一个必须正视的问题。

1. 流程之前

笔者在过往的团队管理中，曾经发现这样一个有趣的现象，组员 A 向 leader 反应，最近团队中有一个非常不好的现象发生，会留下很多 technical debt，leader 听了之后勃然大怒，什么，这还得了，必须整治，于是一个新的流程产生了，各种动员大会，雄心勃勃，这下不稳定因素可以彻底杜绝了。但事实上，各组员破口大骂，什么玩意啊，架构不改进，搞啥流程啊，这下做事更麻烦，从而各种消极情绪导致了热情大大降低，反而直接影响团队氛围。

这就是典型的古代皇帝管理模式，而不是真实理解故事背后的起因和背景，直接搞个政策打压各种民间奇巧淫技，殊不知产生这种现象是对于现状的无奈之举。因此 CMDB 也是如此，在直接跳入流程这个漩涡之前，笔者希望团队完全明白流程产生的故事背景，甚至从工作实践中共同改进流程的方式来一起推动 CMDB，并且得益于 CMDB。

(1) 宏观理解 CMDB

CMDB 的定义广为人知，即配置管理数据库 (Configuration Management DataBase)，可是真正的宏观理解是根据自己项目的实际，以及本项目的 CMDB 的实现方法，来理解当初该软件背后的设计理念和期待解决的问题。

以下是配置项 (CI) 分层。

- ❑ 基础架构层，如 rack 位置信息，硬件型号。
- ❑ 系统层，如 os 版本，安装的软件。
- ❑ 业务层，如游戏还是商城，代码版本。

以下是 CI 来源。

- ❑ rack 位置是手动收集，由 DC 组现场人员录入，硬件型号是 mini ISO 自动收集录入，由 DC 组负责。

- ❑ os 版本和安装的软件是由系统管理员指定, cobbler 和 Puppet 读取安装, 由 SA 组负责。
- ❑ 游戏还是商城, 代码版本, 是由业务运维人员指定, Puppet 读取安装, 由业务组负责。

当理解了分层和来源后, 相当于整个团队都明确了, 当信息缺失或者不准确的时候应该找谁, 哪些环节是容易出错的, 比如手动收集部分, 如果要变更, 谁可以帮忙进行审查, 这样互相协作的时候更透明, 团队氛围也更趋向于互相理解, 而不是各种踢皮球抱怨。

(2) 处处使用 CMDB

花费大力气建设 CMDB 的原因就是最终在日常工作中得益于它, 因此, 各团队都要尊重其权威性, 工作的展开以其为中心, 比如上文提到的机房容量评估, 在使用 openDCIM 后, DC 组的工作简直是事半功倍, 而上文 Puppet 的 enc 脚本接入, 也进一步体现出 SA 组合业务组处处使用 CMDB 的真谛, 让一切生产上的配置项都是通过 CMDB 的信息产生。

(3) 积极反馈 CMDB

笔者团队中, 曾经有这样一个故事, 那时候 CMDB 系统刚上线, 是由一个北美团队实现的, 由于项目初期, 系统还不够完善, 软件中的 worker 经常卡死, 但是由于时区问题, 交流只靠邮件, 各种抱怨。事实证明, 消极的情绪就像瘟疫一样蔓延的飞快, 于是不高兴写自动化任务, 用本地 cache file 来解决问题, 甚至有人扬言要重起炉灶。后来笔者与上级经理直接介入, 与远程团队电话会议, 制定 SLA, 甚至飞到当地直接进行一些反馈以及设计改进的工作。最终, 软件稳定性问题解决了, 并且按照实际面临的问题进行改进, 团队评价高了, 相应的工作开展起来也顺利多了。

因此, 积极反馈, 还要加上实施团队的积极配合, 才能做到一款用户体验友好的 CMDB, 毕竟当今互联网的一个重要理念就是用户体验至上, 项目制造的产品如此, 内部工具也理应如此, 毕竟 CMDB 要赢得的所有的市场(所有攻城狮), 而不是 20% 就是胜利。

(4) 共同进化 CMDB

共同进化意味着 CMDB 的使用者和设计者密不可分, 比如使用者应该占据设计决策的重要位置, 而设计者必须也是使用者, 才能体会用户的真实感受。比如设计者就是 DevOps, 平时也不完全脱离 ops, 只是一半左右的时间在维护 CMDB 上, 另外一半时间他化身为运维, 在各个场景穿梭, 尝试使用自己设计的 CMDB。

(5) 再次迭代

经过上述几步, 肯定不断有新的版本产生, 或者随着线上业务架构的不断进化, CMDB 本身的设计或者已经过于陈旧需要重构, 比如 Cloud 化、Docker 化。但这些都不意味着, CMDB 设计之初的失败, 全体成员需要适应不停地迭代, 重新学习, 反馈, 再进化的过程。就像业务的架构随着业务扩张演变一样, CMDB 也是一个演变的过程, 而每个演变历程, 各个团队都是其主导者, 因为 CMDB 的成功并不是一个 DevOps 组的成功, 而是全团队对于规范化、集中化、自动化管理线上业务的成功实践。

2. 制定流程

在上一节里，提到了 CMDB 的分层和信息来源，细心的读者肯定也会注意到，一个单一的流程并不能满足所有的场景，比如 DC 组的、SA 组的、业务组的，而且每个组内部可能还会细分，比如 DC 组上架的 CMDB 使用流程，DC 组维修的 CMDB 使用流程，因此，这里只是举一个例子，不同大小的公司肯定有不同的组织架构，不能一概而论，适合自己的项目的才是最好的。比如，SA 组升级 Nginx 软件的流程，可以分为如下步骤。

- 1) 从 CMDB 中查询现有 Nginx 版本。
- 2) 通过一台机器测试升级过程，以 Puppet 进行控制。
- 3) 先只更改 game_dev 环境的 Nginx 版本，通过 project_env level 的 CI 项控制版本。
- 4) 版本化管理 CI 项，git push 的时候，指定 peer review，比如 game 业务组。
- 5) peer review 通过，验证 game_dev 升级是否顺利。
- 6) 通知所有业务组，给出变更细节（一个 git pull 的 request 地址，可以用开源的 gerrit 或者 gitlab，也可以自己买 github 私有 repo），确认 test 环境和 prod 时间，每推进一个环境进行 Peer review，git push 到所有业务。

从这个流程可以看出，这个流程涉及到以下几个重点。

- ❑ 利用 git 来管理 CI 的版本和 review 工作，中心化管理变更。
- ❑ 团队协作以 CMDB 的变更为中心进行沟通确认，如 pull request 地址。
- ❑ 全体团队都是依照 dev → test → prod 的最佳实践进行变更。

整个流程都穿插了技术准则，有非常强的可操作性，而不是纯粹管理人员订的不知所云的空话大话，这便是一个成功流程制定的范例，必定来自生活，还原于生活。

11.6.2 CMDB 与自动化

说起自动化，很多人肯定会说自动化啊，交给 DevOps，或者 ops tooling 吧，这是那个小组的事情。殊不知自动化和 CMDB 一样是一种相辅相成的理念，CMDB 的主要应用场景就是自动化，而自动化成功的前提就是完善的 CMDB，要想 CMDB 的成功，离不开团队里每一个人对于自动化和 CMDB 的信仰。而团队里不需要每个人都是 Python 高手，但是需要每个人都至少会 shell 来调用 CMDB 的接口，从而写一些简单的自动化脚本，哪怕只是一个监控相关，或者助力运维的一个小脚本。

举个简单的例子，新同事小 A 是一个新手运维，对于公司的 CMDB 也是停留在表面理解，没有真正调用过，他在平时的运维过程中有如下几个弱点。

- ❑ 使用别的同事写的工具的时候，一定需要别人的协助，排错。
- ❑ 很多时候，在完成任务的时候，使用了最原始的方法，不懂得写一些可以让事半功倍的小脚本，导致效率低下。
- ❑ 特别容易抱怨这个工具不好用，那个系统（CMDB）不靠谱，其他团队（如 DevOps 团队）和他讨论问题，经常觉得在鸡同鸭讲，口碑非常不好。

这是一个典型的新手运维的样板，如果不进行一些改变的话，他就会成为传说中的团队毒瘤，自身职业发展也会受到限制。

在其 team leader 谆谆教导下，小 A 认识到自己的不足，开始从工作点滴中开始写一些自动化的小程序，由于程序需要，他仔细研究了 CMDB 的调用，以及其分层，了解了其中的理念，他深深感到之前由于自己的无知，不仅浪费同事们那么多时间帮助他，而且还对于之前因为不理解 CMDB 调用的复杂度，而乱喷其他小伙伴感到羞愧，后来他积极地回馈，参与改进 CMDB 以及各种自动化脚本的完善工作，也得到了团队的信任和认可，在年度绩效考核中也名列前茅。

这其实是一个很典型的团队故事，在 CMDB 的推行过程中，和时下推行 DevOps 的理念其实是一致，在一个团队中越多人理解并参与其中，这个团队就越能体会到 CMDB 和 DevOps 所带来的无与伦比的优势。

11.6.3 做好 CMDB 的架构设计

CMDB 的重要性已经不言而喻了，而面对如此重要的一环，良好的架构设计是不可或缺的，否则用户体验将大打折扣，如同糟糕的项目会失去市场份额一样，非健壮的 CMDB 架构最终会失去内部的民心。

1. 可扩展性

这一点，DevOps 人员最深有体会，在写了一个自认为牛逼闪闪的脚本后，雄心勃勃要批量运行时，发现调用的 CMDB 各种不响应、超时，然后有不知情的同事抱怨你的脚本不好用时，那真的欲哭无泪。因此，性能的需求是 CMDB 架构设计中非常重要的一项，当然每种 CMDB 的实现方式不一样，因此扩展方式都不一样，比如 openDCIM，是典型的 PHP+MySQL 的应用。

前端 PHP+Apache (Nginx)，完全可靠加机器解决。后端 MySQL，如果只有读的压力，可以用 slave 解决，并指定相应只读前端，如果写有压力的话，可以尝试 percona-server-cluster。其他的 CMDB，如是自建的可以尝试使用 queue、异步、DB 分片等高级扩展技术。

2. 高可用性

这点想必所有人都深有体会，特别是负责搭建的人体长假的时候，那真是自上而下的抓狂，各种加班熬夜，最后通牒，务必在下个工作日前恢复上线，不然所有配套的系统都无法运行，所有部门效率直接减去 80%。因此高可用性对于 CMDB 来说，是非常高的优先级的。同样，具体怎么实施，不是本节的重点，不做展开。

3. 稳定性

其实稳定性是把双刃剑，如果 CMDB 的 API 只是时不时地会报错，但是可用率还是在 90% 以上，也是对写工具人的一种考验，比如引导其写好 retry 机制、error handler、

timeout 机制等，这些反而都是不错的导向，总比平时不出错，到关键时候发现一出错就彻底崩溃的客户端工具要可靠得多。在笔者的项目中，CMDB 的作者甚至有一个配置选项是设定百分之多少的出错率，来强迫客户端工具在撰写的过程中就发现不稳定，从而写好各种 retry 机制、error handler、timeout 机制，笔者在看到这个 idea 之后，深深拜服。

当然还有一种稳定性是致命的，比如给出错误的答案，曾经笔者项目有一次 CMDB 卡死，不知怎么返回空字典，写 DNS 自动化的同事也没有写判断，从而整个 DNS A 记录被清空，然后各种灾难产生。

因此，后者的那种稳定性是 CMDB 里最致命的，在 CMDB 的服务端和客户端都应该积极规避这样的情况发生。

4. 可追溯性

CMDB 的可追溯性分为 2 种：

- ❑ 用户行为的追溯性，又称审计。审计的重要性，应该是众所周知的，不再赘述，就算再简陋的 CMDB，也可以尝试嵌入 git 进行配置修改，从而得益于 git 本身的审计功能。
- ❑ 系统行为的追溯性，又称可以 traceback 的 log。系统行为，不仅仅是为了方便 crash 的 troubleshooting，也为了上文给出错误答案稳定性的情况，提供一个可追溯的手段，否则到时候连原因都不明，从而不知道如何修复，时时刻刻都会是一颗定时炸弹。

11.6.4 那些年，我们碰过的坑

笔者在工作期间有幸看到初期 CMDB 的搭建一直到今天，其中的酸甜苦辣都尝了个遍，分享下那些年碰过的坑，与各位读者共勉，也希望各位读者能尽早跳出各种火坑，真正把 CMDB 由自己掌控，犹如手握绝世好剑，驰骋在运维的沙场。

坑一，开发者和使用者没有交集

当年刚做 CMDB 的时候，一群国外有志青年，兴致勃勃用 Django 做了一套 CMDB 的雏形，基础的信息都有，如 hostname、dc、project、env、ip、puppet_role，全公司一致叫好，于是乎各种新的 idea 层出不穷，比如给开发一套 DB 管理程序，给 GM 一套游戏管理界面，还有接入 metrics、整合 log，等等，忙得不亦乐乎。但是随着时间的推移，DevOps 用户发现 API 并不完善，只有按照 hostname 的 search，没有其他 search 项，比如 ip 的 API 非常麻烦，由于有多 ip 的功能，因此需要解开一层列表的循环才能拿到主 ip，但是生产上没人用这个功能，仅仅一些个人测试机才需要这个功能，只是开发者当初设计时的个人机需要，造就了无数复杂的自动化脚本。项目发展到最后，重点都在怎样和业务结合，做更多的开发需求，而忽视了简化 ops 工作这个初衷。最终，公司另外一群有志青年 DevOps 另起炉灶，做了一套满足需求的，把 Django 的那套 inventory 功能彻底替换掉了，当然用了月余，

才把老的自动化工作全迁移到新 CMDB 上。

这个坑是典型的开发者和使用者没有交集，从而导致了：

- ❑ 做出来的和想要的不一样。
- ❑ CMDB 发展方向并不是痛点。
- ❑ 没有反馈，也没有收集。

坑二，没有按照应用场景进行设计

第二套系统上架后，那些写自动化脚本的同事们一片叫好，各种舒心的 search API、filter API 一应俱全，于是乎，又是各种需求蜂拥而至，做 LB status 的接入，switch status 的接入，以及 puppet kick 的集成。但是用户却觉得功能虽然非常多，却在一些常见场景中，有这样那样的问题，比如部署新版本的时候，从 dev → test → prod，经常是 dev 迭代 10 个版本，test 才迭代 3 个，而 prod 才 1 个，而所有配置内容都是以 key value 的形式零散地存在 CMDB 中，于是出现了在 test 和 prod 部署的时候漏掉 conf change，需要人工的形式 tag 这些 conf change 到下次 Puppet 变更中的情况。最终，公司一位少年发怒了，做了一套基于 git 的 conf management center，以 CMDB 为存储位置，做了这些 key value 的 tagging 到 version 的工作。

这个坑是典型的想当然开发，并没有按照实际应用场景进行设计。做出来的可以解决问题，但需要曲线救国。仅仅停留在最最基础的 key value 存储，对于写基于 CMDB 做自动化工作的人来说，不要像乐高玩具一样，给用户一堆零散的零件，让其自己一块一块地搭起来，耗时多，又容易出错，用户体验很差，只有耐心非常好又对 CMDB 有深刻理解的用户才能正确使用。所以设计者应该以场景为导向设计模块，比如用户要搭一个马里奥世界，与其提供颜色相近的积木，不如提供相应的怪兽、马里奥、公主以及一些道具的常用模块，会让用户更容易上手，用户体验自然大大加分。

坑三，忽略用户体验

conf management center 做好后，为了有 API，又为了结合 git，做了一个所谓的 json patch，旨在纯粹用 json patch 这个 feature，把其他格式的配置文件全部转化为 json，如 yam->json、ini->json，想法不错，这样不同环境之间的迭代，只需要打 patch 即可。但是，由于格式之间的转化有损，需要加一系列 Ruby 的 Puppet function 来修正，比如 value 类型修复，再转回 yaml 要 sort_yaml 等，把 Puppet 代码搞得非常复杂，充斥着这些 custom function。最终，笔者团队的一位组员发奋图强，做了一套 UI，隐藏了这些复杂的逻辑。

这个坑是典型的 geek 开发的工具，没有像一个对外产品设计一样，做足用户体验的功课。

- ❑ 导致用户体验极差，新用户失去学习使用的欲望
- ❑ 排错的时候，设置了一定的障碍
- ❑ 使用的时候，也增加了出错的可能性

从上述三个坑可以看出，笔者的 CMDB 项目也是在演变中，出现各式各样的问题，不同阶段有不同阶段所需要面对的挑战，这些经验教训是一笔宝贵的财富，目前笔者的项目在用无盘系统，可以做到重启后，在没有硬盘的情况下，通过 Puppet+CMDB，ramdisk 里的新系统和老系统一模一样，这是项目初期无法想象的事情。

希望各位读者在各自的 CMDB 不要怕遇到坑，只要克服后停下脚步总结、自审、再演变，相信都可以做出一套适合自己项目的 CMDB，make life easier！

日志管理

12.1 日志中的四个 W

日志是系统管理员的排错利器，也是程序员调试分析程序的必要工具，一条规范的日志中总是会包含四个 W，分别是 When、Where、Who 和 What。

- ☐ When: 事件是何时发生的。
- ☐ Where: 日志是在哪里产生。
- ☐ Who: 哪个程序触发了这条日志。
- ☐ What: 发生了什么事，以及事件的内容。

很多情况下，系统管理员会忽视对日志的管理，当需要追查问题的时候却发现因为对日志的管理不当，使得查找问题变得复杂且低效。本章将重点讲解如何管理与分析日志，让日志变得更有效，让分析排除更为高效。

12.2 首先要有一个日志服务器

通常来说，日志会记录在本地服务器上，或者是交换机、路由器等的网络设备的内部存储中，但是这样不够灵活，当需要排除时，系统管理员或网络管理员需要登录每台服务器、每个网络设备去查看日志，这在面对 10 多台设备时，管理员一般还能承受，但如果有一千台设备，这显然是非常低效的。所以我们需要一台日志服务器去集中化地收集存储日志，方便系统管理员、程序员查看日志，方便存储更久时间的日志做分析。

图 12-1 是一个日志收集系统的基本架构，网络设备、服务器通过 syslog 的协议将日志

传送到日志服务器上，日志服务器定时地将日志归档，存储到后端存储设备中。

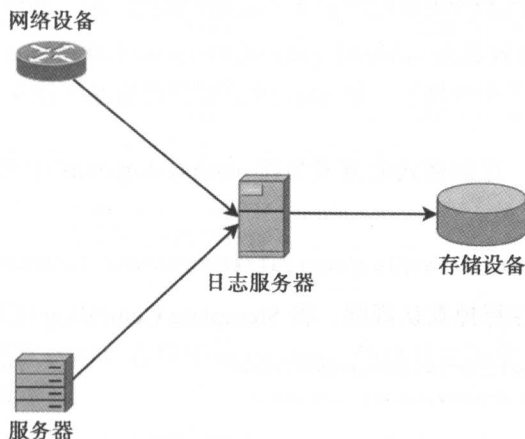


图 12-1 日志收集系统

常用的日志管理程序为 rsyslog 和 syslog-ng，这两个日志管理程序都支持下面三种日志传输方式：

- ❑ UDP 传输：这是应用最广泛的日志传输方式，性能开销小，但是传输缺乏可靠性。
- ❑ TCP 传输：确保日志传输的可靠性，但是性能开销大于 UDP 方式。
- ❑ TLS 加密传输：确保日志传输的可靠性与安全性。

12.2.1 rsyslog

在 CentOS 6 中，rsyslog 作为系统默认的日志管理程序，版本为 v5-stable，在 EPEL 源中提供了 v7-stable 版本。在生产环境中如果没有特殊需求，推荐使用系统默认版本。如果对日志传输有性能的要求，推荐使用 EPEL 源中的 v7-stable 版本。当然，如果读者是版本追新者，也可以从 rsyslog 官网（www.rsyslog.com）上下载最新的 v8-stable。

1. rsyslog 服务端配置

服务端比较简单，只需要配置传输协议、端口、存储位置即可。

首先，打开 /etc/rsyslog.conf 中 udp、tcp 514 端口：

```
$ModLoad imudp
$UDPServerRun 514
```

```
$ModLoad imtcp
$InputTCPServerRun 514
```

然后，重启 rsyslog 服务，此时会看到 udp、tcp 514 端口都已被监听，代码如下：

```
[root@rsyslog ~]# /etc/init.d/rsyslog restart
Shutting down system logger: [ OK ]
Starting system logger: [ OK ]
```

```
[root@rsyslog ~]# netstat -nltp | grep 514
tcp        0      0 0.0.0.0:514          0.0.0.0:*            LISTEN      1562/rsyslogd
tcp        0      0 :::514              :::*                  LISTEN      1562/rsyslogd
[root@rsyslog ~]# netstat -nlup | grep 514
udp        0      0 0.0.0.0:514          0.0.0.0:*            1562/rsyslogd
udp        0      0 :::514              :::*                  1562/rsyslogd
```

配置日志存储路径、存储格式的方式为在 `/etc/rsyslog.conf` 中加入如下行，即可定义日志存储的位置以及方式。

```
$template Centrallog, "/var/log/central/%HOSTNAME%/%PROGRAMNAME%.log"
```

配置日志规则时，注释掉默认规则，将 `$template Centrallog` 应用在这条规则上。

```
*.info;mail.none;authpriv.none;cron.none    /var/log/messages
*.info;mail.none;authpriv.none;cron.none    ?Centrallog
```

最后，重启 `rsyslog` 服务，此时 `rsyslog` 服务器配置完成，`/var/log/central` 中就会自动出现本机 log。

2. rsyslog 客户端配置

在 `/etc/rsyslog.conf` 中加入下面两行，可以自行选择使用 UDP 方式传输 log 还是 TCP 方式。如果两者都打开，在 `rsyslog` 服务上会看到同一条 log 发送了两次。使用 UDP 传输 log 时用一个 `@`，用 tcp 方式传输 log 时用两个 `@@`，添加完成之后重启 client 端的 `rsyslog` 服务即可。

```
# UDP方式传输log
*. *@rsyslog.example.com:514

# TCP方式传输log
*. *@@rsyslog.example.com:514
```

下面测试是否配置成功，在 `rsyslog` 客户端上用 `logger` 命令生成一条日志，如下：

```
[root@es01 ~]# logger "hello"
```

此时 `rsyslog` 服务器上就会存储下这条 log，如下：

```
[root@rsyslog ~]# tail -f /var/log/central/es01/root.log
May  2 17:34:49 es01 root: hello
```

OK，基于 `rsyslog` 的集中化日志服务器配置成功。

12.2.2 syslog-ng

`syslog-ng` 是另外一种流行的日志管理的解决方案，它是一个商业软件，包括收费版和开源版两种版本，EPEL 源提供了 `syslog-ng` 开源版本的 RPM 安装包，官方网站中提供源码包，可以选择从官方网站下载源码编译安装，也可以使用 EPEL 中的 RPM 包，此处使用 EPEL 源安装 `syslog-ng`。

1. 启用 EPEL 源

如果服务器可以通外网，则可以将 `baseurl` 直接指向外网的 EPEL 镜像地址，比如 sohu 源地址 `http://mirrors.sohu.com/fedora-epel/6Server/x86_64`，或者将 EPEL 源同步至内网，建立自己的 yum 源，这里采用了笔者自己建立的 yum 源，代码如下：

```
[root@syslog-ng ~]# cat /etc/yum.repos.d/epel.repo
[epel]
name=Extra Package For Enterprise Linux
baseurl=http://192.168.0.254/repo/6/epel
enable=1
```

2. 安装 syslog-ng

因为在 CentOS 6 中默认的日志程序是 `rsyslog`，所以这里需要先删除 `rsyslog` 程序，然后再安装 `syslog-ng` 程序。

```
[root@syslog-ng ~]# yum remove rsyslog
[root@syslog-ng ~]# yum install syslog-ng
```

3. 配置 syslog-ng 服务器端

相对来说，`syslog-ng` 的配置文件看起来要比 `rsyslog` 复杂了许多，但是如果能够理解配置含义就会发现其实并不难配置，大致分为四步骤。

首先做全局配置，这里有几个参数需要注意。

- ❑ `flush_lines`：设置一次发送多少条日志，默认为 0，也就是收到日志就发出。
- ❑ `use_dns`：是否使用 DNS，其中有个一个参数是 `persist_only`，这个参数是只从 `/etc/hosts` 中解析主机名，不依赖 DNS 服务器。
- ❑ `log_msg_size(10240)`：设置单条日志大小，超过括号内的数值就会被丢弃，如果不设置，默认值为 8192 bytes。有的时候，程序员可能为了 debug，会将应用程序的报错信息全部塞进一条日志的 `message` 部分，这可能会导致单条日志很大，所以这里需要根据实际情况去调整参数值。
- ❑ `create_dirs`：如果存储日志的位置是以层级目录的形式存在的，则需要打开这个参数，让 `syslog-ng` 自动创建需要的目录。

全局配置如下：

```
options {
    flush_lines (0);
    time_reopen (10);
    log_fifo_size (1000);
    long_hostnames (off);
    use_dns (no);
    use_fqdn (no);
    create_dirs (yes);
    keep_hostname (yes);
};
```

然后设置日志来源、端口、协议、最大连接数，以及存储的位置，如下：

```
source demo_source {
    tcp(ip(0.0.0.0) port(514) max-connections(1000));
};
destination d_central { file("/var/log/central/$HOST/$PROGRAM"); };
```

syslog-ng 在这里给了很多的变量，以便设置存储方式，如 \$HOST 日志来源主机、\$PROGRAM 生成日志的程序、\$YEAR \$MONTH \$DAY \$HOURL \$SEC 时间变量等。

完成前两步之后，就需要设置日志的过滤器了。一般来说，可以不用设置过滤器，但是遇到一些特殊情况，比如想把含义某些特殊字符的日志过滤出来，放入一些特定的位置，那么过滤器会是一个很好的帮手。

例如，要将负载均衡器日志中含有 Deny 字样的日志过滤出来，写法如下：

```
filter f_loadbalance { host(lb.example.com) and match("Deny" value("MESSAGE"))};
```

最后，设置规则，将来源 demo_source 中的日志存储到 d_central 中：

```
log { source(demo_source); destination(d_central); };
```

这样服务器端配置完成，重启 syslog-ng 服务让配置生效。

4. 客户端配置

客户端的配置较为简单，只要加入目标位置与规则，重启 syslog-ng 让配置生效即可：

```
destination d_syslog { tcp("syslog-ng", port(514));};
log { source(s_sys); destination(d_syslog);};
```

此时就可以在服务器端看到客户端传送的日志了。

12.2.3 如何选择 syslog 程序

常见的 syslog 程序主要包括 rsyslog 和 syslog-ng，但其实远不止这些软件，比如老牌的 syslog、Windows 系统上事件查看器、facebook 开源的 scribe 等，但是相对来说，rsyslog 和 syslog-ng 两者在性能、开发进度，以及各个平台的支持程度上最为突出。个人实际经验中，rsyslog 的性能略好于 syslog-ng，且 rsyslog 作为 CentOS 默认的日志程序，也免去了一些安装配置的工作，个人推荐尽量使用 rsyslog。

同时推荐阅读 syslog 的 RFC 文档 (<http://www.ietf.org/rfc/rfc3164.txt>)，对理解 syslog 协议会有很大的帮助。

12.3 常见的日志分析处理工具

当用户拥有了一个集中日志服务器时，在存储上睡大觉的日志不会产生任何价值，只

有对其进行分析才能进一步发挥日志的价值。例如公司的安全部门想知道在过去的一段时间内有多少次的 ssh 登录失败、来源是哪里,以及有哪些账户登录失败,这些需求就需要对日志进行分析统计,最终整理出报表交付给相关部门。

对日志的分析有很多种方式,可以采用自己写脚本提取日志内容,也可以采用一些开源或者商业工具来自动分析,现在有许多这样的工具可供选择。下面会介绍常见的日志分析程序。

1. Splunk

Splunk 是一款极其强大的可视化日志分析软件,它号称是日志届的 google,在大数据与云计算日趋流行的今天,Splunk 被认为是监控体系中不可缺少的一个重要软件。通过快速分析日志来达到与传统监控软件不一样的报警。Splunk 更为关注整体监控,更容易得到趋势分析。它的网址为 <http://www.splunk.com/>。

Splunk 可以高效地对日志进行索引、分析,并生成报表。它是一个商业软件,但同时也提供了免费试用版,试用版每天只能索引 500MB 日志。后面会讲述如何安装配置 Splunk。

2. Loggly

Loggly 是由前 Splunk 员工创立的,它是构建在 Amazon AWS 上的一个第三方的云日志处理平台,它将日志存储在云端,并提供分析处理能力。Loggly 提供了更简单的日志管理分析方式。这对刚刚起步的小型企业是非常节省成本的,只需要将日志导入 Loggly 平台即可近乎实时地获取日志分析结果,快速高效。但是缺点也明显,在高流量高并发的日志环境中,Loggly 做得还不够好。同时因为构建在 AWS 上,对网络带宽要求也较高。Loggly 的网址为 <https://www.loggly.com/>。

3. Logstash

有商业软件就有开源软件,Logstash 就是一个强大的开源日志分析软件,Logstash、Elasticsearch 与 Kibana 组成目前最流行的开源日志处理平台。

Logstash 可以说是一个日志处理的过程框架,由 input、filter、output 三个部分组成,在这个框架上,开源社区提供了 100 多个插件,几乎涵盖了所有能遇到的日志处理情况。

Logstash 现在属于 Elasticsearch 公司 (<https://www.elastic.co>),网址为 <http://logstash.net>。本章的后面会重点讲解如何配置 Logstash+Elasticsearch+Kibana。

4. garylog2

garylog2 也是一个非常优秀的日志聚合软件,它与 Logstash 几乎同时出现在日志处理的舞台上,早先 garylog2 的后端存储是基于 MongoDB 的,因为 MongoDB 在当时出现的一些问题,使得很多人望而退步,错过了与 Logstash 竞争的机会。它的网址为: <https://www.graylog.org/>。

12.4 Splunk 的安装配置

Splunk 支持多种平台, 包括 Windows、Linux、Mac OS, 以及 Solaris, 本节将讲述在 CentOS 6 上安装配置 Splunk, 使用 splunk 对日志进行分析。

12.4.1 下载 Splunk 安装程序包

打开 Splunk 的下载页面 http://www.splunk.com/en_us/download.html, 选择 Splunk Enterprise 进入版本选择界面, 然后选择需要的平台安装包, 这里选择 Linux。

从图 12-2 中可以看到, 在 Linux 平台上, Splunk 是以支持的内核版本形式下载的, 而不是以某个发行版的形式发布, 可见 Splunk 对平台兼容性考虑得很周到。

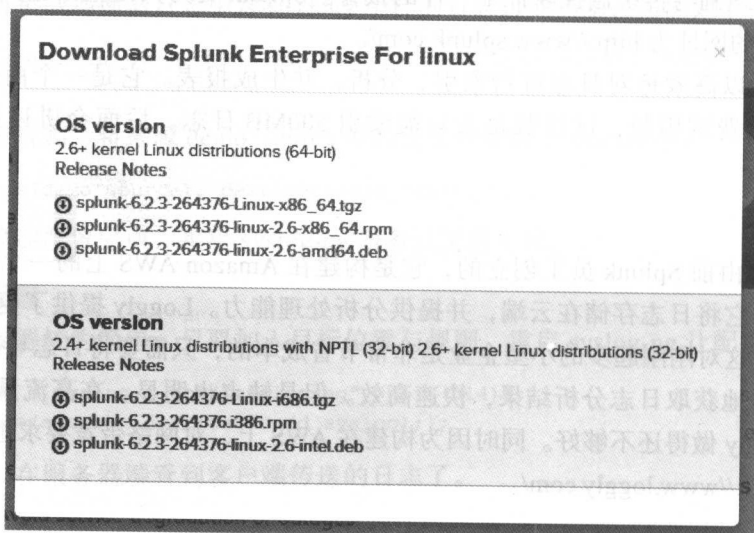


图 12-2 Splunk 支持形式

这里选择 `splunk-6.2.3-264376-linux-2.6-x86_64.rpm` 下载, 随后进入下载界面, Splunk 需要注册一个账户后才能获得下载地址, 这里不再重复账户注册过程。

12.4.2 安装启动 Splunk

下载完成之后将 RPM 包拷贝至服务器上, 安装过程非常简单, 一个命令就完成了。这里安装在 `syslog-ng` 服务器上, 命令如下:

```
[root@syslog-ng ~]# rpm -ivh splunk-6.2.3-264376-linux-2.6-x86_64.rpm
warning: splunk-6.2.3-264376-linux-2.6-x86_64.rpm: Header V3 DSA/SHA1 Signature,
key ID 653fb112: NOKEY
Preparing... ##### [100%]
1:splunk ##### [100%]
complete
```


启动 Splunk 的方式:

```
[root@syslog-ng ~]# /opt/splunk/bin/splunk start
```

Splunk 首次启动时会会有一个 LICENSE AGREEMENT, 将空格拉到底表示同意即可, 图 12-3 是许可证的截图。

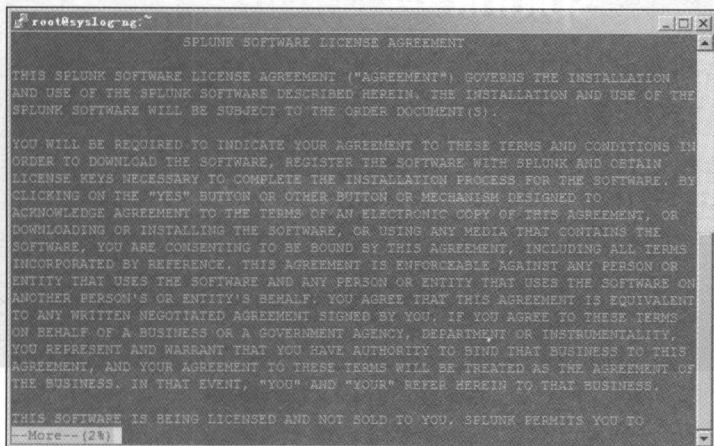


图 12-3 Splunk LICENSE AGREEMENT

待 Splunk 启动完成之后, 打开 Web 浏览器访问 Splunk 服务器的 8000 端口, 图 12-4 是 Splunk 的启动过程。

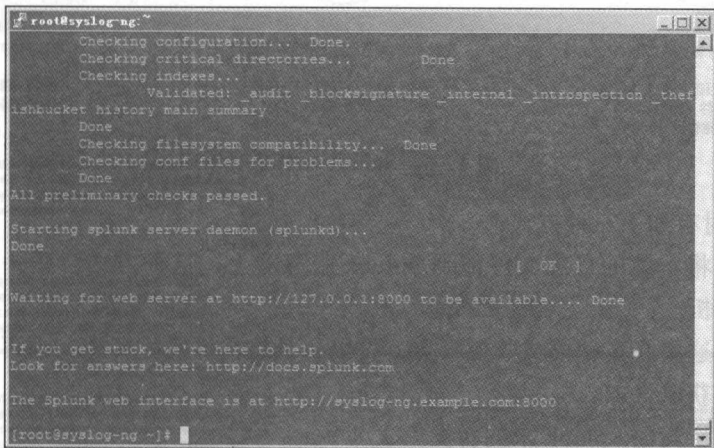


图 12-4 Splunk 启动过程

12.4.3 配置 Splunk

首次登录 Splunk 的时候, 默认的用户名为 Admin, 密码为 changeme, Splunk 会提示用户更改为自己的密码。图 12-5 是 Splunk 首次登录的状态。

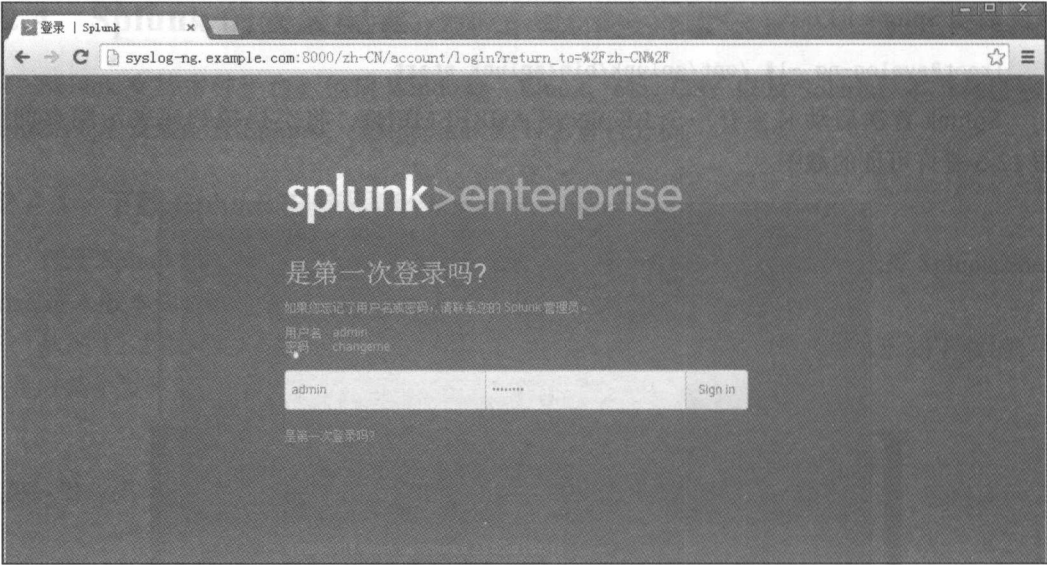


图 12-5 Splunk 初始登录

登录之后，就可以开始配置数据源了。选择添加数据，如图 12-6 所示。

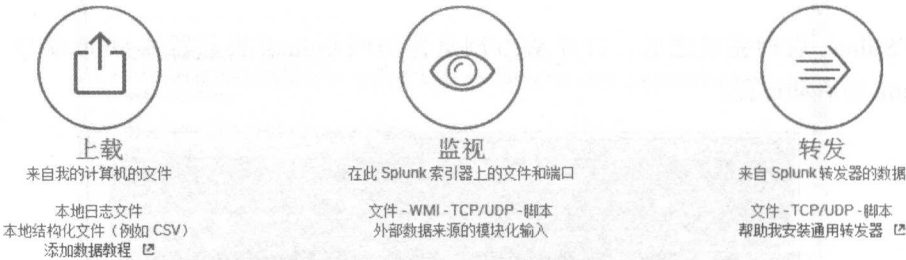


图 12-6 Splunk 添加数据源

添加数据时有如下三种方式：

- ❑ 上载：将本地的日志文件传到 Splunk 服务器上，这种方式合适一些非实时数据的离线分析。
- ❑ 监视：这里有多种模式监控日志，可以从日志文件监控日志，也可以从 syslog 的端口直接获取日志。
- ❑ 转发：从其他 Splunk 来的转发。

这里使用监视作为输入源，选择文件和目录形式，需要配置监控的目录，这里考虑监控 /var/log 这个目录，请看图 12-7，然后点击下一步。

在输入设置这里，需要格外注意主机的配置。不同的日志里，Host 这个字段可能处在不同的位置，正常情况下是第三段，这里选择使用“路径中的段”，段号是 3，请看图 12-8。然后点击“检查”进入下一步。

添加数据

选择来源 输入设置 检查 完成

文件和目录
 上传文件、编制本地文件的索引或监视整个目录。

TCP / UDP
 配置 Splunk，以侦听网络端口。

脚本
 使用脚本从任何 API、服务或数据库中获取数据。

Configure this instance to monitor files and directories for data. To monitor all objects in a directory, select the directory. Splunk monitors and assigns a single source type to all objects within the directory. This might cause problems if there are different object types or data sources in the directory. To assign multiple source types to objects in the same directory, configure individual data inputs for those objects. [了解更多信息](#)

将跳过数据预览，其不支持目录。

文件或目录? [浏览](#)

在 Windows 上: c:\apache\apache.error.log 或 \\hostname\apache\apache.error.log。在 Unix 上: /var/log 或 /mnt/www01/var/log。

白名单?

黑名单?

常见问题

- > Splunk 可索引何种类型的文件?
- > 我无法访问我要索引的文件。为什么?
- > 我如何在 Splunk 实例上获取远程数据?
- > 除了内容外，我还可以监视文件的更改吗?
- > 来源类型是什么?
- > 我该如何为目录指定白名单或黑名单?

图 12-7 设置 Splunk 监控目录

添加数据

选择来源 输入设置 检查 完成

输入设置
 可根据需要将这些数据的其他输入参数设置如下:

Sourcetype
 The source type is one of the default fields that Splunk assigns to all incoming data. It tells Splunk what kind of data you've got, so that Splunk can format the data intelligently during indexing. And it's a way to categorize your data, so that you can search it easily.

自动 选择 手动

应用上下文
 Application contexts are folders within a Splunk instance that contain configurations for a specific use case or domain of data. App contexts improve manageability of input and source type definitions. Splunk loads all app contexts based on precedence rules. [了解更多信息](#)

应用上下文

主机
 When Splunk indexes data, each event receives a "host" value. The host value should be the name of the machine from which the event originates, and can be defined based either on the path to the source data, a regular expression, or a number that represents a segment of a file path. [了解更多信息](#)

常量值 路径的正则表达式 路径中的段

段号?

索引
 Splunk stores incoming data as events in the selected index. Consider using a "sandbox" index as a destination if you have problems determining a source type for your data. A sandbox index lets you troubleshoot your configuration without impacting production indexes. You can always change this setting later. [了解更多信息](#)

索引 [创建新索引](#)

图 12-8 主机字段设置

如果检查通过，就可以看到图 12-9 的界面，可以开始搜索了。

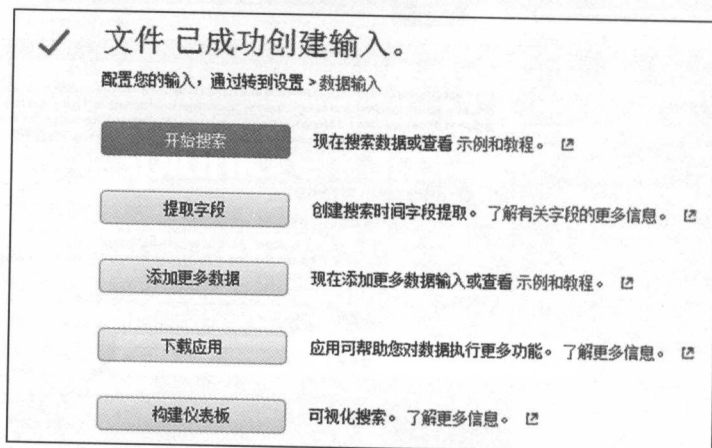


图 12-9 Splunk 输入源配置

12.4.4 搜索日志

Splunk 搜索的语法很简单，只需输入想知道的关键字就可以，同时也支持正则表达式。图 12-10 是在 Splunk 里搜索 ssh 登录日志。

12.5 Elasticsearch+Logstash+Kiana

12.5.1 ELK 简介

纵然 Splunk 简单方便，但因高昂的售价使得用户不得不转向廉价的解决方案，而开源的 Elasticsearch、Logstash 与 Kibana 组成了现在最流行的 ELK 日志分析系统。

在 ELK 体系中，log 的处理流程如下。

log -----> Logstash -----> Elasticsearch -----> Kibana

当一条 log 产生以后，发送给 Logstash，Logstash 对这条 log 进行字段解析，解析完成之后，它将其存入 Elasticsearch 中，最后用户使用 kibana 来查询这条 log。

Logstash 是日志解析工具，它处理日志分为三步：输入→解析→输出，而这每一步都是由众多插件组成的，所以 Logstash 可以做到“Ship logs from any source, parse them, get the right timestamp, index them, and search them”。具体插件可以查阅 Logstash 的官方文档来获取 (<http://logstash.net/docs/1.4.2/>)。图 12-11 是 Logstash 支持的所有插件。

Elasticsearch 是一个实时的全文检索和分析引擎，在行业内获得了非常高的认可度，GitHub、维基百科等网站都使用了 Elasticsearch 作为内部检索平台，优秀的检索能力也非常适合日志的查询分析。

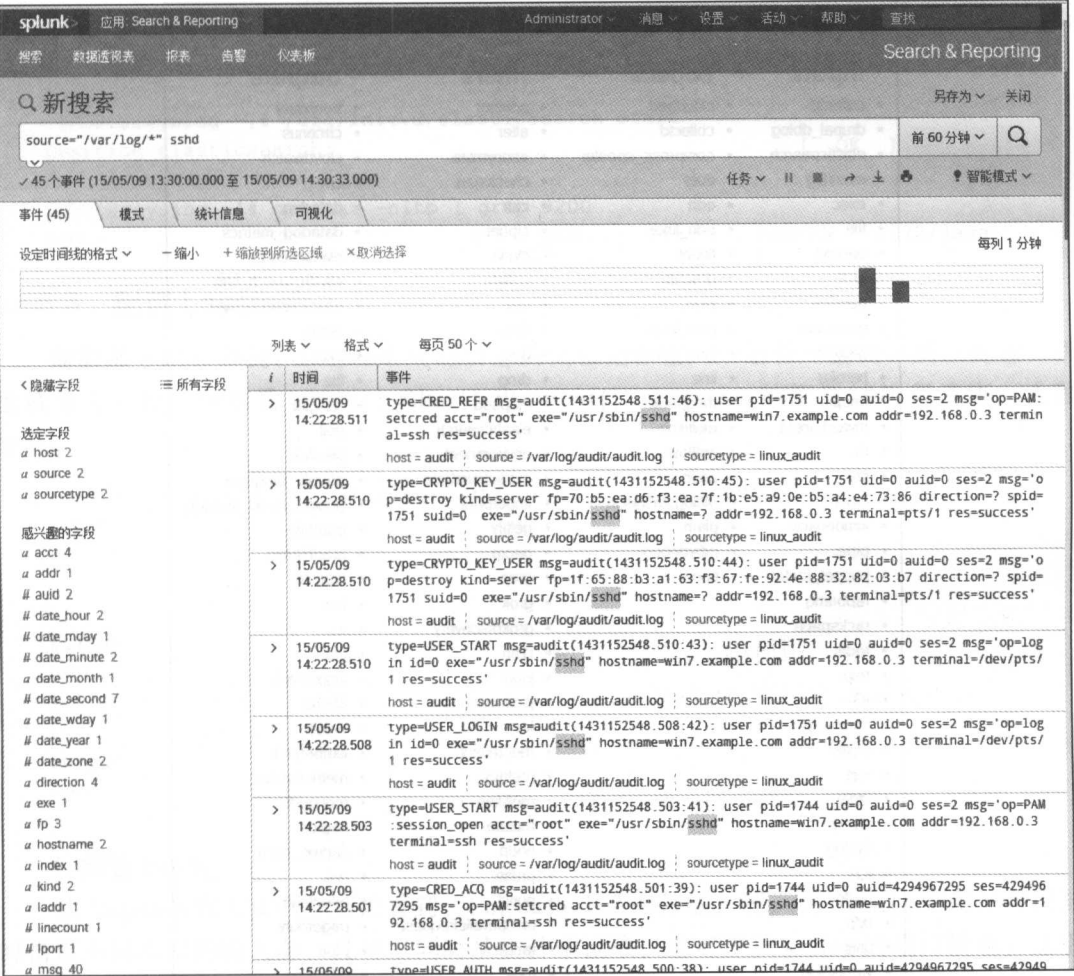


图 12-10 Splunk 搜索 ssh 登录日志

Kibana 是日志展示系统，通过 Kibana 可以搜索查询日志，Kibana 的开发也非常活跃，从第一版到现在的第四版，每一版作者都对其进行了重构。但是也正是因为不断的重构，给用户带来了不小的不适应。

12.5.2 安装 ELK 软件包

1. 下载软件包

Logstash 与 Kibana 已经成为了 Elasticsearch 官方产品，只需要到 Elasticsearch 的官方网站（<https://www.elastic.co/>）下载即可。其中 Logstash 需要下载 Logstash 与 logstash-contrib 两个包，它们分别是 logstash 的主程序与 Logstash 的插件。

plugin documentation			
inputs	codecs	filters	outputs
<ul style="list-style-type: none">• collectd• drupal_dblog• elasticsearch• eventlog• exec• file• ganglia• gelf• gemfire• generator• graphite• heroku• imap• invalid_input• irc• jmx• log4j• lumberjack• pipe• puppet_factor• rabbitmq• rackspace• redis• relp• s3• snmptrap• sqlite• sqs• stdin• stomp• syslog• tcp• twitter• udp• unix• varnishlog• websocket	<ul style="list-style-type: none">• cloudfail• collectd• compress_spooler• dots• edn• edn_lines• fluent• graphite• json• json_lines• json_spooler• line• msgpack• multiline• netflow• noop• oldlogstashjson• plain• rubydebug• spool	<ul style="list-style-type: none">• advisor• alter• anonymize• checksum• cidr• cipher• clone• collate• csv• date• dns• drop• elapsed• elasticsearch• environment• extractnumbers• fingerprint• gelfify• geoip• grep• grok• grokdiscovery• i18n• json• json_encode• kv• metaevent• metrics• multiline• mutate• noop• prune• punct• railsparallelrequest• range• ruby• sleep	<ul style="list-style-type: none">• boundary• circonus• cloudwatch• csv• datadog• datadog_metrics• elasticsearch• elasticsearch_http• elasticsearch_river• email• exec• file• ganglia• gelf• gemfire• google_bigquery• google_cloud_storage• graphite• graphtastic• hipchat• http• irc• jira• juggernaut• librato• loggly• lumberjack• metriccatcher• mongodb• nagios• nagios_nsca• null• opentsdb• pagerduty• pipe• rabbitmq• rackspace

图 12-11 Logstash 支持的插件

2. 安装 Logstash 与 Elasticsearch

Logstash 和 Elasticsearch 都需要依赖 Java，所以首先要安装 Java，命令如下：

```
[root@rsyslog ~]# yum install java-1.7.0-openjdk.x86_64
```

安装 Logstash 的命令如下：

```
[root@rsyslog ~]# rpm -ivh logstash-1.4.2-1_2c0f5a1.noarch.rpm
[root@rsyslog ~]# rpm -ivh logstash-contrib-1.4.2-1_efd53ef.noarch.rpm
```

安装 Elasticsearch 的命令如下：

```
[root@rsyslog ~]# rpm -ivh elasticsearch-1.5.2.noarch.rpm
```

安装完 Logstash 和 Elasticsearch 之后，先启动 Elasticsearch，当看到 9200 端口被监听的时候，说明 Elasticsearch 已经正常启动，然后进入 Logstash 的配置阶段。

```
[root@rsyslog ~]# /etc/init.d/elasticsearch start
Starting elasticsearch: [ OK ]

[root@rsyslog ~]# netstat -nltp | grep 9200
tcp                0          0 :::9200          :::*             LISTEN            1772/java
```

12.5.3 配置 Logstash

前面部分已经说到，Logstash 处理日志的流程是：输入→解析→输出，所以配置文件也就分为三段，分别是 input{}、filter{} 和 output{}。Logstash 的配置结构很清晰，大致如下：

```
[root@rsyslog ~]# cat /etc/logstash/conf.d/logstash.conf
Input {
.....
}

filter {
.....
}

output {
.....
}
```

1. 配置 input{}

从 Logstash 官方文档上可以看到，Logstash 支持多种输入源，在 input{} 中用户可以同时指定不同类型的输入源，且可以同时从文件中读取，也可以从 syslog 的端口读取，这里选择使用文件作为输入源。配置命令如下：

```
input {
  file {
    path => ["/var/log/central/*/*"]
    type => "syslog"
    start_position => "beginning"
  }
}
```

- ❑ path：定义日志文件路径，path 在这里其实指的是一个列表，如果你需要指定多个路径，写法是 path => ["/var/log/message", "/var/log/mail"]。
- ❑ type：定义日志类型，这里是由用户自己决定类型名称，目的是为下一步解析配置中能够配置相对应的解析方式。
- ❑ start_position：指的是从文件开头开始读取，还是从文件的末尾开始读取。

2. 配置 filter{}

相对来说, filter{} 的配置略微复杂, 尤其是其中的 grok 部分, 需要使用辅助工具帮助调试, 下面先看 filter{} 部分的配置:

```
filter {
  if [type] == "syslog" {
    grok {
      match => { "message" => "%{SYSLOGTIMESTAMP:syslog_timestamp} %{SYSLOG-
        HOST:syslog_hostname} %{DATA:syslog_program} (?:\[%{POSINT:syslog_
        pid}\])?: %{GREEDYDATA:syslog_message}" }
      add_field => [ "received_at", "%{@timestamp}" ]
      add_field => [ "received_from", "%{host}" ]
    }
    syslog_pri { }
    date {
      match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd HH:mm:ss" ]
    }
  }
}
```

可以看到, filter 的第一行是以 if[type] 开头的, 这里的 type 就是在 input{} 中对 log 的定义, 既然是 if 语句, 那必然支持 else。

grok 可以说是整个 Logstash 的精华部分, 通过 grok 这个插件可以帮助用户自定义日志格式解析, 也就是说无论是多复杂的日志, grok 都可以帮按照用户需求解析出来。这里以系统日志为例, 系统日志由时间、hostname、program、信息四个部分组成, 在 grok 中, 每一个字段用 %{} 来定义, 而有些日志在 program 部分带着 pid, 那就是用 %{}() 的方式将 program 和 pid 分开。

grok 的编写有时候会比较痛苦, 因为不同的需求会有不同的写法, 尤其在需要正则表达式匹配的地方, 好在有 grok debug 这个在线工具可以帮助编写 grok (<https://grokdebug.herokuapp.com/>)。

3. 配置 output{}

output{} 的配置比较简单, 指定 Elasticsearch 的 host 和 cluster 即可。配置命令如下:

```
output {
  elasticsearch {
    host => localhost
    cluster => "example"
  }
}
```

12.5.4 配置 Elasticsearch

在 Elasticsearch 中需要配置 cluster.name, 因为默认情况 Elasticsearch 集群是通过广播方式进行通信的, 指定不同的 cluster.name 可防止干扰内网中其他 Elasticsearch 集群。配置

命令如下：

```
[root@rsyslog ~]# cat /etc/elasticsearch/elasticsearch.yml | grep cluster.name
cluster.name: example
```

12.5.5 配置 Kibana

修改 kibana4 配置文件中 `elasticsearch_url` 的地址就可以让 Kibana 连接上 Elasticsearch。默认情况下 Kibana 端口是 5601，这里为了实验方便，将 port 指向了 80 端口，生产环境中建议使用 Apache proxy 将 80 指向到 5601 的端口。相关命令如下：

```
[root@rsyslog kibana-4.0.2-linux-x64]# cat config/kibana.yml | grep -v ^# | grep -v ^$
port: 80
host: "0.0.0.0"
elasticsearch_url: "http://localhost:9200"
elasticsearch_preserve_host: true
kibana_index: ".kibana"
default_app_id: "discover"
request_timeout: 300000
shard_timeout: 0
verify_ssl: true
bundled_plugin_ids:
- plugins/dashboard/index
- plugins/discover/index
- plugins/doc/index
- plugins/kibana/index
- plugins/markdown_vis/index
- plugins/metric_vis/index
- plugins/settings/index
- plugins/table_vis/index
- plugins/vis_types/index
- plugins/visualize/index
```

然后手动启动 kibana4，命令如下：

```
[root@rsyslog kibana-4.0.2-linux-x64]# bin/kibana
{"@timestamp":"2015-05-09T17:22:40.125Z","level":"info","message":"Found kibana
  index","node_env":"production"}
{"@timestamp":"2015-05-09T17:22:40.140Z","level":"info","message":"Listening on
  0.0.0.0:80","node_env":"production"}
```

现在可以开始配置 Kibana 了。首次打开 Kibana 之后，会让用户配置默认的 index pattern，只需要选择 `@timestamp` 之后点击 `create` 就完成了基础配置，如图 12-12 所示。

完成初始化配置之后，会看到 Kibana 已经获取到了 Elasticsearch 里的存储内容，如图 12-13 所示。

点击图 12-14 左上角的 `discover`，选择需要查询的时间段，便可以查询到需要的日志内容。

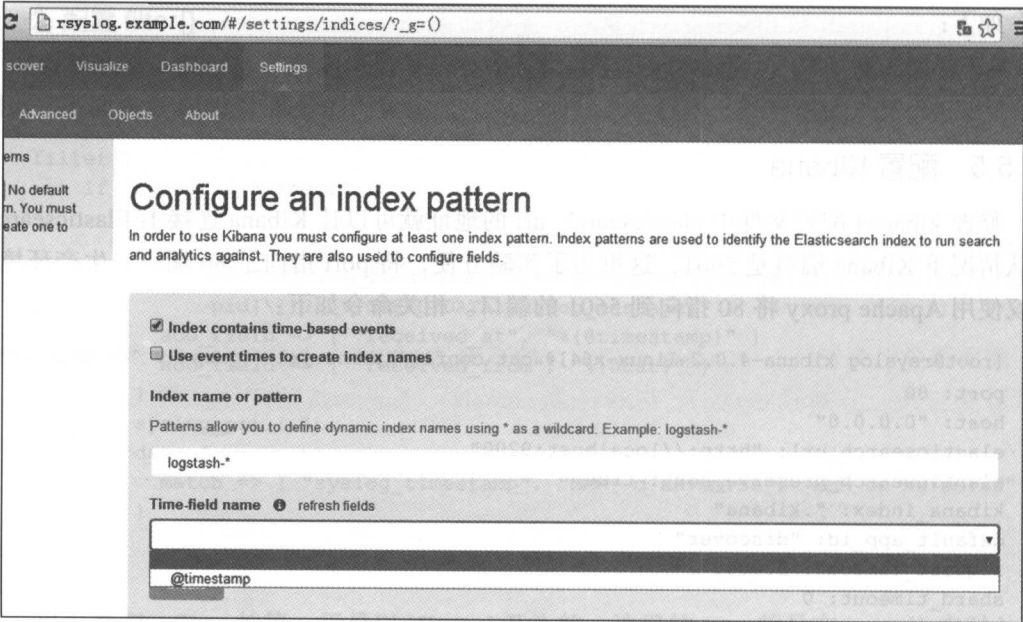


图 12-12 Kibana 初始配置

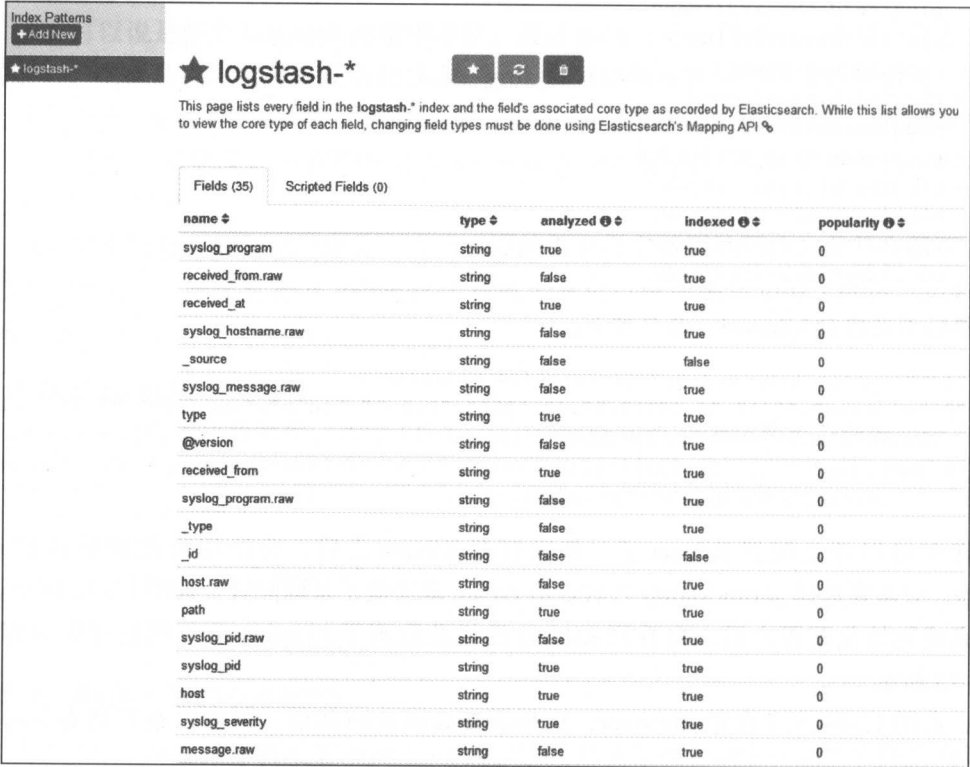


图 12-13 Kibana 连接 Elasticsearch



图 12-14 Kibana 查询日志

12.6 Elasticsearch 入门

Elasticsearch 是一个强大的全文检索分析引擎，在 ELK 体系中，Elasticsearch 是非常重要的一个环节，本节会简单讲解 Elasticsearch 的基本配置、插件安装、API 调用等内容。

12.6.1 基本配置

1. Elasticsearch 服务配置

在 /etc/elasticsearch 中有两个配置文件，分别是 elasticsearch.yml 和 logging.yml。elastic-

search.yml 文件用于负责 Elasticsearch 运行参数的配置, logging.yml 的文件定义了 Elasticsearch 服务本身的日志信息, 如果需要获取更详细的 Elasticsearch 日志, 就需要对 logging.yml 里的一些文件参数做修改。

在 elasticsearch.yml 里, 有一部分参数是可以在运行的时候通过 API 来修改的, 而有一部分参数则无法通过 API 修改, 必须在配置文件中修改, 尤其是两个重要的值 cluster.name 和 node.name。

Cluster.name 是一个 Elasticsearch 集群的名称, Elasticsearch 启动的时候通过广播方式找到其他 Elasticsearch 节点, 然后通过判断 cluster.name 来决定是不是属于同一个 cluster, 如果相同, 则加入集群。

Node.name 是 Elasticsearch 在加入集群时的标记, 即自己的名字, 默认可以不更改, Elasticsearch 会自动为自己取一个唯一的名称。命令如下:

```
[root@es01 ~]# cat /etc/elasticsearch/elasticsearch.yml | grep -v ^# | grep -v ^$
cluster.name: example
node.name: es01
```

2. 配置集群的 master 节点和 data 节点

Elasticsearch 集群使用 zen discovery 方式形成集群, 它提供了多播和单播两种发现方式, 默认采用以多播方式组成集群。当 Elasticsearch 集群启动的时候, 默认第一个启动的 Elasticsearch 是 master 节点, 而 Elasticsearch 的默认配置既是 master 节点, 同时也是 data 节点。

当 Elasticsearch 集群很大 (有几十台) 的时候, 配置比较好的方式是指定某台 Elasticsearch 成为固定的 master 节点, 但这台 master 节点并不存储数据, 只负责管理整个集群, 其他 Elasticsearch 则成为固定的 data 节点, 配置过程如下。

在 master 的节点的 elasticsearch.yml 中进行如下配置:

```
node.master: true
node.data: false
```

在 data 的节点 elasticsearch.yml 中进行如下配置:

```
node.master: false
node.data: true
```

同时注意, 不同版本的 Elasticsearch 不要混用组成集群。

3. 配置 Elasticsearch JVM 内存

Elasticsearch 默认启动的 JVM 内存限制为 1024MB, 这个默认值在实际情况中毫无可用性, Elasticsearch 官方建议将 JVM 的内存限制在主机内存的 50% 以内, 最大不要超过 32GB, 给操作系统和其他程序留下足够的内存, 以供使用。同时, 建议将 ES_MIN_MEM 和 ES_MAX_MEM 两个设置为相等的值, 以减少内存换入换出损失 Elasticsearch 性能。

其交互，最简单的方式就是使用 curl 来和 Elasticsearch 进行交互。比如，想知道一共存了多少条日志，就可以使用下面的这种方式获取。

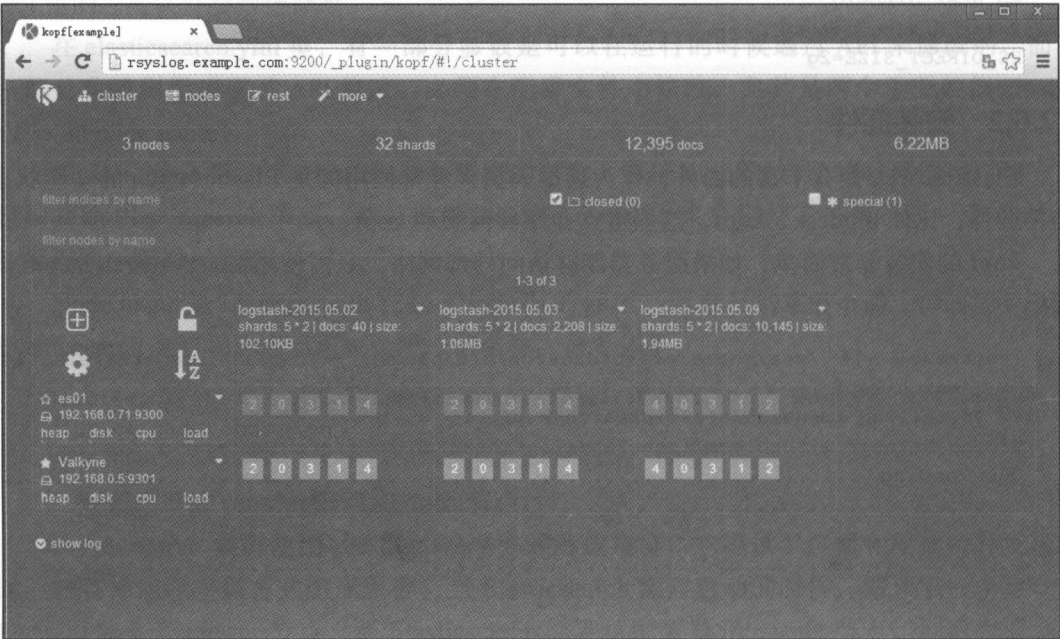


图 12-15 Kopf 状态图

```
[root@es01 ~]# curl -XGET 'http://localhost:9200/_count?pretty' -d '{
> {
>   "query": {
>     "match_all": {}
>   }
> }
> '
{
  "count" : 12395,
  "_shards" : {
    "total" : 16,
    "successful" : 16,
    "failed" : 0
  }
}
```

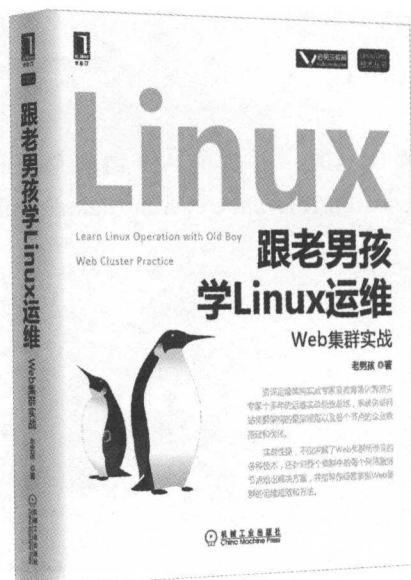
对于系统管理员来说，常用的 API 莫过于 health 了，使用 health API 可以快速地看到 Elasticsearch 的状态。命令如下：

```
[root@es01 ~]# curl -XGET localhost:9200/_cluster/health?pretty
{
  "cluster_name" : "example",
  "status" : "yellow",
```

```
"timed_out" : false,  
"number_of_nodes" : 1,  
"number_of_data_nodes" : 1,  
"active_primary_shards" : 16,  
"active_shards" : 16,  
"relocating_shards" : 0,  
"initializing_shards" : 0,  
"unassigned_shards" : 16,  
"number_of_pending_tasks" : 0  
}
```

Elasticsearch 有非常全面的 API，更多的 API 请查阅 Elasticsearch 官方手册 <https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html>。

推荐阅读



跟老男孩学linux运维：web集群实战

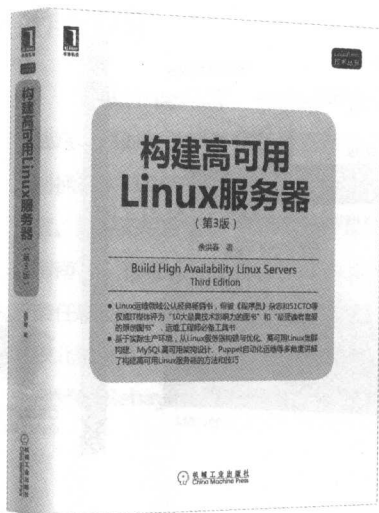
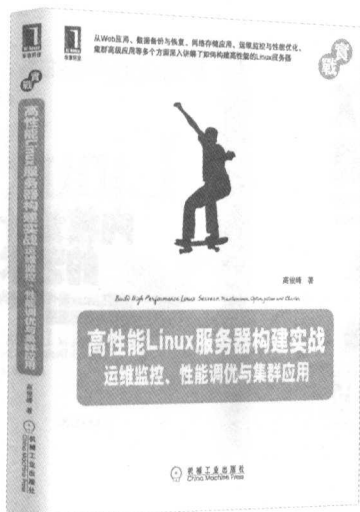
书号：978-7-111-52983-5 作者：老男孩 定价：99.00元

**资深运维架构实战专家及教育培训界顶尖专家十多年的运维实战经验总结，
系统讲解网站集群架构的框架模型以及各个节点的企业级搭建和优化**

本书不仅讲解了Web集群所涉及的各种技术，还针对整个集群中的每个网络服务节点给出解决方案，并指导你细致掌握Web集群的运维规范和方法，实战性强。

互联网运维涉及的知识面非常广，本书涵盖了构架一个Web网站集群所需要的基础知识，以及常用的Web集群开源软件使用实践。通过本书的实战指导，能够帮助新人很快上手搭建一个完整的Web集群架构网站，并掌握相关的知识点，从而胜任企业的运维工作。

推荐阅读



高性能Linux服务器构建实战：运维监控、性能调优与集群应用

作者：高俊峰 ISBN: 978-7-111-36695-9 定价：79.00元

毫无疑问，Linux服务器是企业级服务系统的主流，随着企业各种数据量的不断增加，企业对服务器系统可靠性、稳定性方面的要求越来越高，越来越突出，高可靠性、高稳定性已经成为评价业务系统性能的主要指标。影响Linux服务器系统性能的因素有很多，改善Linux服务器系统性能的方法和工具也很多，本书紧紧围绕“高性能”这个话题，从Web应用系统、数据备份恢复、网络存储、运维监控、性能优化、集群应用等多方面讲解了构建高性能Linux服务器系统的方法和最佳实践，其中性能优化和集群应用这两个话题是本书的重点。

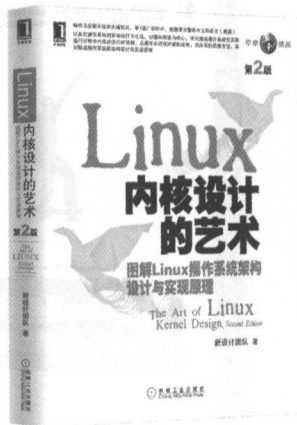
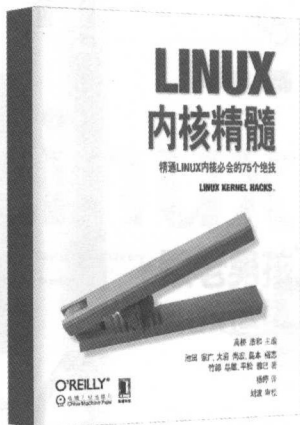
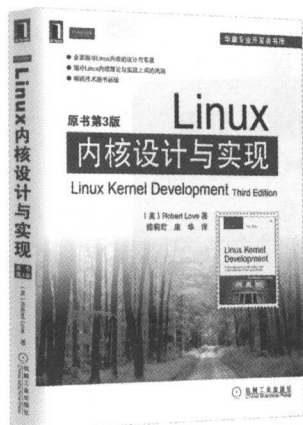
构建高可用Linux服务器 (第3版)

作者：余洪春 ISBN: 978-7-111-47787-7 定价：79.00元

本书自第1版出版以来，就广受关注和好评曾被《程序员》杂志和51CTO等权威IT媒体评为“10大最具技术影响力的图书”和“最受读者喜爱的原创图书”，作者根据运维技术的发展和读者的反馈意见，不断地对书的内容进行优化：更新了过时的技术和方法；补充了最新的内容；限于篇幅，部分内容作为电子版免费提供给读者下载；使得这本书的内容更加完善。

本书最大的特点就是与实践紧密结合，所有理论知识、方法、技巧和案例都来自实际生产环境，涵盖Linux服务器构建与优化、服务器故障诊断与排除、Shell脚本、高可用Linux集群构建、MySQL性能调优及高可用、自动化运维(Puppet)、安全运维等主题，所有内容都围绕“如何构建高可用的Linux服务器”这个主题深度展开。

推荐阅读



Linux内核设计与实现（原书第3版）

作者：Robert Love ISBN：978-7-111-33829-1 定价：69.00元

本书基于Linux 2.6.34内核详细介绍了Linux内核系统，覆盖了从核心内核系统的应用到内核设计与实现等各方面内容。主要内容包括：进程管理、进程调度、时间管理和定时器、系统调用接口、内存寻址、内存管理和页缓存、VFS、内核同步以及调试技术等。同时本书也涵盖了Linux 2.6内核中颇具特色的内容，包括CFS调度程序、抢占式内核、块I/O层以及I/O调度程序等。本书采用理论与实践相结合的路线，能够带领读者快速走进Linux内核世界，真正开发内核代码。

Linux内核精髓：精通Linux内核必会的75个绝技

作者：Hirokazu Takahashi 等 ISBN：978-7-111-41049-2 定价：79.00元

本书从先进Linux内核的众多功能中选取了一些基本而且有趣的内容进行介绍，同时也对内部的运行和结构进行了讲述。此外还介绍了熟练使用这些功能所需的工具、设置方法以及调整方法等。本书还为想要了解Linux内核的读者以及读过本书后开始对Linux内核开发产生兴趣的读者，介绍了获取内核源代码的方法和内核开发方法等内核构建入门所需的信息。

Linux内核设计的艺术：图解Linux操作系统架构设计与实现原理（第2版）

作者：新设计团队 ISBN：978-7-111-42176-4 定价：89.00元

本书的特点在于，既不是空泛地讲理论，也不是单纯地从语法的角度去逐行地分析源代码，而是以操作系统在实际运行中的几个经典事件为主线，将理论和实际结合在一起，精准地再现了操作系统在实际运行中究竟是如何运转的。宏观上，大家可以领略Linux 0.11内核的设计指导思想，可以了解到各个环节是如何牵制并保持平衡的，以及软件和硬件之间是如何互相依赖、互相促进的；微观上，大家可以看到每一个细节的实现方式和其中的精妙之处。

作者简介

本书的作者王军、林荣、潘野、徐洲、张黎明在撰写本书时都就职于著名游戏制作公司动视暴雪，分别拥有8~10年的工作积累，理论和实战经验极其丰富。其中王军曾出版《Linux系统命令及Shell脚本实践指南》，有着极好的读者评价和赞誉。成书之时，林荣任职动视暴雪上海运维团队主管；王军、张黎明任职北美暴雪高级工程师；徐洲任职HSBC高级工程师；潘野任职ebay高级工程师。

随着互联网技术的迅猛发展，互联网应用已经触及并深入到人们的日常生活中，为人们提供了极大的便利，Linux作为被广泛使用的各类应用的后端操作系统，也因此迎来了重要的大规模应用，Linux集群服务应运而生。那么，如何更简易、更快捷、更有效地管理Linux集群呢？本书作者将以自己的实践经验为基础，通过实战案例讲解大规模Linux集群服务的应用与运维技巧。

本书的主要内容和特色：

- 结合大量的实践案例，以授之以渔的方式对大规模Linux集群架构技术点进行总结和归纳，从而迅速提高读者的实战能力。
- 5位作者都是各自负责领域的专家，对开源工具的利用和理解有着深厚的功力和独到的见解，本书取众人之所长，更好地保证了内容的准确性与实用性。
- 从多个技术角度详尽地描述了Linux集群的设计、搭建、部署和运维，特别强化了自动化、规模化的概念，帮助读者快速上手并掌握相关知识。
- 随着Linux开源软件的发展，涌现了一批非常好的自动化备份、配置管理、日志分析等工具，本书针对这一系列工具进行了深入讲解。
- 以鼓励动手的方式增强阅读效果，读者在多次阅读后技能上会有本质的提高。



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/Linux

ISBN 978-7-111-57585-6



9 787111 575856

定价: 79.00元